# Optimal Partitioning for Dual Pivot Quicksort[⋆]

Martin Aumüller and Martin Dietzfelbinger

Faculty of Computer Science and Automation, Ilmenau University of Technology, 98694 Ilmenau, Germany
martin.aumueller@tu-ilmenau.de, martin.dietzfelbinger@tu-ilmenau.de

**Abstract.** *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 lower order or linear terms. 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. We also present results about minimizing the average number of swaps.

## 1 Introduction

Quicksort [7] is a thoroughly analyzed classical sorting algorithm, described in standard textbooks such as [3,8,12] 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, those smaller than the pivot and those larger than the pivot, and 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 $2n \ln n + O(n)$ comparisons. In 2009, Yaroslavskiy announced[1] 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 $p < q$ are used, the

[⋆] An extended abstract of this work has been presented at *ICALP*'13.

[1] 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 [15].

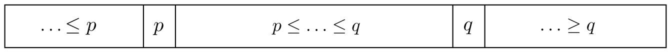

**Fig. 1.** Result of the partition step in dual pivot quicksort schemes using two pivots $p, q$ with $p \le q$. All elements $\le p$ are moved to the left of $p$; all elements $\ge q$ are moved to the right of $q$. All other elements lie between $p$ and $q$.

partitioning step partitions the remaining $n - 2$ elements into 3 parts: those smaller than $p$ (*small* elements), those in between $p$ and $q$ (*medium* elements), and those larger than $q$ (*large* elements), see Fig. 1.[2] Recursion is then applied to the three parts. As remarked in [15], it came as a surprise that two pivots should help, since in his thesis [10] Sedgewick had proposed and analyzed a dual pivot approach that was inferior to classical quicksort. Later, Hennequin in his thesis [6] studied the general approach of using $k \ge 1$ pivot elements. According to [15], he found only slight improvements that would not compensate for the more involved partitioning procedure. (See [15] for a short discussion.)

In [15], Nebel and Wild formulated and analyzed a simplified version of Yaroslavskiy's algorithm. (For completeness, this algorithm is given as Algorithm 5 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 paper, also following tradition, we concentrate on the comparison count as cost measure and on asymptotic results.

The authors of [15] state that the reason for Yaroslavskiy's algorithm being superior were 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 behavior. In this paper 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 w.r.t. the average comparison count and identify optimal members (up to lower order or linear terms) 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 [6] the connection between partitioning cost and overall cost for quicksort variants with more than one pivot is analyzed in detail. The result relevant for us is that if two pivots are used and the (average) partitioning cost for $n$ elements

[2] For ease of discussion 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 [11].

can be bounded by $a \cdot n + O(1)$, for a constant $a$, then the average cost for sorting $n$ elements is

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

Throughout the present paper 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 [15] 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 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 that is influenced by a particular partitioning procedure.

In the paper, we will 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 $\ell/(n-2)$, resp., 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 $s$ [$m$, $\ell$] small [medium, large] elements, and there is no independence. Still, we will show that the randomness in the order is sufficient to guarantee that

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

The details of the partitioning procedure will determine $f^{\mathrm{p}}_{s,\ell}$ and $f^{\mathrm{q}}_{s,\ell}$, and hence $w_{s,\ell}$ up to $o(n)$. 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 lower order terms determine $f^{\mathrm{p}}_{s,\ell}$ and $f^{\mathrm{q}}_{s,\ell}$ for this partitioning procedure. This will give $w_{s,\ell}$ up to lower order terms, which must then be averaged over all $s, \ell$ to find the average number of comparisons in partitioning. Then apply (1).

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

We shall demonstrate approach (i) in Section 4. An example: As explained in [15], if $s$ and $\ell$ are fixed, in Yaroslavskiy's algorithm we have $f^{\mathrm{q}}_{s,\ell} \approx \ell$ and $f^{\mathrm{p}}_{s,\ell} \approx s + m$. By (2) we get $w_{s,\ell} = (\ell s + (s+m)\ell)/n + o(n)$. This must be averaged over all possible values of $s$ and $\ell$. The result is $\frac{1}{4}n + o(n)$, which together with $\frac{4}{3}n + O(1)$ gives $\frac{19}{12}n + o(n)$, close to the result from [15].

Principle (ii) will be used to identify an (asymptotically) optimal partitioning procedure that makes $1.8n \ln n + o(n \ln n)$ 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 ($f^{\mathrm{p}}_{s,\ell} \approx n$ and $f^{\mathrm{q}}_{s,\ell} \approx 0$), otherwise compare (almost) all entries with the larger pivot first ($f^{\mathrm{p}}_{s,\ell} \approx 0$ and $f^{\mathrm{q}}_{s,\ell} \approx n$). 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 will consider the following simple and intuitive strategy: *Compare the current element to the smaller pivot first if more small elements than large elements have been seen so far, otherwise compare it to the larger pivot first.* We will show that this is optimal w.r.t. the average comparison count with an error term of only $O(n)$ instead of $o(n \ln n)$.

In implementations of quicksort, the pivot is usually not chosen directly from the input. Instead, one takes a sample of three elements from the input and chooses the median of these elements as the pivot. Theoretically, this decreases the average comparison count from $2n \ln n + O(n)$ to $1.714n \ln n + O(n)$. In Section 7 we will analyze the comparison count of dual pivot quicksort algorithms that use the tertiles in a sample of 5 elements as the two pivots. Yaroslavskiy's algorithm has an average comparison count of $1.704n \ln n + O(n)$ in this case, while the optimal average cost is $1.623n \ln n + O(n)$. At the end of that section we will compare the average comparison count of classical quicksort and optimal dual pivot quicksort algorithms for some sample sizes. The result is surprising: when both algorithms use the same sample size, classical quicksort beats (optimal variants of) dual pivot quicksort even at such small sample sizes as 5.

In order to arrive at a full dual pivot quicksort algorithm we must also take into account the work for moving elements around. Elements can be moved around by exchanging two elements in the input, a so-called *swap* operation. In Section 8, we will point out the obvious connection between this "moving issue" in dual pivot quicksort and the so-called "dutch national flag" problem, in which an input of elements—each having either the color "blue", "white", or "red"—must be re-arranged such that they resemble the national flag of the Netherlands. In this turn, we will identify strategies for moving elements around that can be used in dual pivot quicksort algorithms, too. We will also state a simple lower bound on the average number of swaps needed to find a partition of the input as depicted in Figure 1.

As noted by Wild *et al.* [18], considering only key comparisons and swap operations does not suffice for evaluating the practicability of sorting algorithms. In Section 9, we will present preliminary experimental results that indicate the following: When sorting integers, the "optimal" method of Section 5 is slower than Yaroslavskiy's algorithm. When making key comparisons artificially expensive, e. g., by sorting strings, we gain a small advantage.

We emphasize that the purpose of this paper 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 $\{1, \ldots, n\}$, 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 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 in between $p$ and $q$ ("medium" elements), and elements larger than $q$ ("large" elements), see Fig. 1. Then apply the procedure recursively to these three groups. Clearly, each pair $p, q$ with $1 \leq p < q \leq n$ appears as 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 [15, 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.* [6,15]) to give

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

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 2 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 $\ell$, we may calculate $\mathrm{E}(P_n)$ as follows:

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

We call the third summand the *additional cost term (ACT)*, as it is the only value that depends on the actual classification algorithm.

## 3 Analyzing the ACT

We will 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 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 $\sigma, \mu, \lambda$, resp., 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.

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

$$c_\pi = \bigl|\bigl\{j \in \{1, \ldots, n-2\} \mid c(e_j) \neq \mu, l(v_j) \neq c(e_j)\bigr\}\bigr|.$$

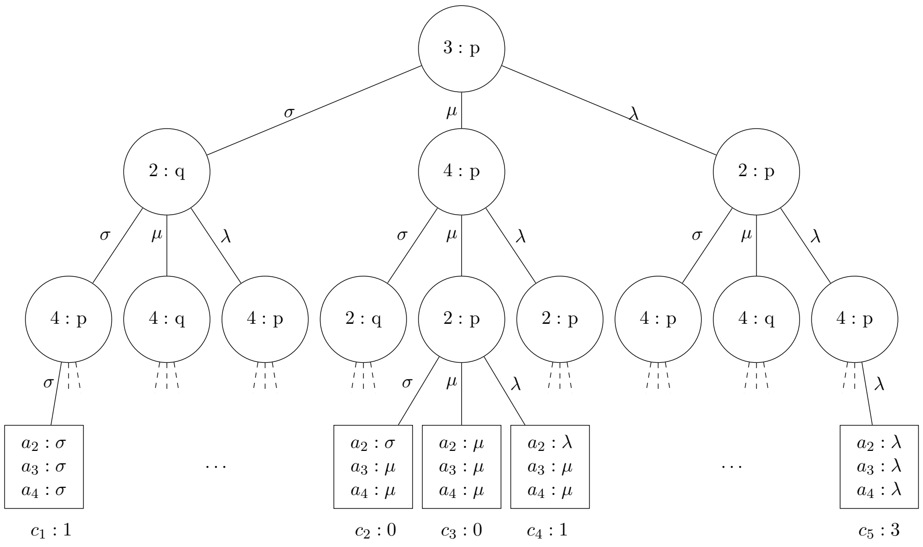
3 : p
σ
μ
λ
2 : q
4 : p
2 : p
4 : p
4 : q
4 : p
2 : q
2 : p
2 : p
4 : p
4 : q
4 : p
$a_2 : \sigma$ $a_3 : \sigma$ $a_4 : \sigma$
$a_2 : \sigma$ $a_3 : \mu$ $a_4 : \mu$
$a_2 : \mu$ $a_3 : \mu$ $a_4 : \mu$
$a_2 : \lambda$ $a_3 : \mu$ $a_4 : \mu$
$a_2 : \lambda$ $a_3 : \lambda$ $a_4 : \lambda$
$c_1 : 1$
$c_2 : 0$
$c_3 : 0$
$c_4 : 1$
$c_5 : 3$

**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.

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 decision tree is given in Figure 2.

We now describe how we can calculate the ACT of a decision tree $T$. Fix $s$ and $\ell$, and let the input excepting the pivots be arranged randomly. For a node $v$ in $T$, we let $s_v$, $m_v$, and $\ell_v$, resp., denote the number of edges labeled $\sigma$, $\mu$, and $\lambda$, resp., 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 $\sigma$ is used, is exactly $(s - s_v)/(n - 2 - \text{level}(v))$. The probability that it is "medium" is $(m - m_v)/(n - 2 - \text{level}(v))$, and that it is "large" is $(\ell - \ell_v)/(n - 2 - \text{level}(v))$. The probability $p^v_{s,\ell}$ that node $v$ in the tree is reached is then just the product of all these edge probabilities on the unique path from the root to $v$. The probability that the edge labeled $\sigma$ 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 $\lambda$ is used is $p^v_{s,\ell} \cdot (\ell - \ell_v)/(n - 2 - \text{level}(v))$. Note that this is independent of the actual ordering in which the decision tree inspects the elements. We can thus always assume some fixed ordering and forget about the label $i(v)$ of a node $v$.

For a random input, we let $S^T_2$ $[L^T_2]$ denote the random variable that counts the number of small [large] elements classified in nodes with label q [p]. By linearity of expectation, we can sum up the contribution to the additional comparison count for each node separately. Thus, we may calculate

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

The setup developed so far makes it possible to describe the connection between a decision tree $T$ and its average comparison count in general. Let $F_\mathrm{p}^T$ resp. $F_\mathrm{q}^T$ be two random variables that denote the number of elements that are compared with the smaller resp. larger pivot first when using $T$. Then let $f_{s,\ell}^\mathrm{q} = \mathrm{E}\left(F_\mathrm{q}^T \mid s,\ell\right)$ resp. $f_{s,\ell}^\mathrm{p} = \mathrm{E}\left(F_\mathrm{p}^T \mid s,\ell\right)$ denote the average number of comparisons with the larger resp. smaller pivot first, given $s$ and $\ell$. Now, if it was decided in each step by independent random experiments with the correct expectations $s/(n-2)$, $m/(n-2)$, and $\ell/(n-2)$, resp., whether an element is small, medium, or large, it would be clear that for example $f_{s,\ell}^\mathrm{q} \cdot s/(n-2)$ is the average number of small elements that are compared with the larger pivot first. The next lemma shows that one can indeed use this intuition in the calculation of the average comparison count, excepting that one gets an additional $o(n)$ term due to the elements tested not being independent.

**Lemma 1.** *Let $T$ be a decision tree. Let $\mathrm{E}(P_n^T)$ be the average number of key comparisons for classifying an input of $n$ elements using $T$. Then*

$$\mathrm{E}(P_n^T) = \frac{4}{3}n + \frac{1}{\binom{n}{2}\cdot(n-2)} \sum_{s+\ell\le n-2} \left(f_{s,\ell}^\mathrm{q}\cdot s + f_{s,\ell}^\mathrm{p}\cdot \ell\right) + o(n).$$

*Proof.* Fix $p$ and $q$ (and thus $s, m$, and $\ell$). We will show that

$$\mathrm{E}(S_2^T + L_2^T \mid s,\ell) = \frac{f_{s,\ell}^\mathrm{q}\cdot s + f_{s,\ell}^\mathrm{p}\cdot\ell}{n-2} + o(n). \tag{7}$$

The lemma then follows by plugging in this result into (5). For a given node $v$ in $T$, we call $v$ *good* if $l(v)=\mathrm{q}$ and

$$\left|\frac{s}{n-2} - \frac{s-s_v}{n-\mathrm{level}(v)-2}\right| \le \frac{1}{n^{1/6}},$$

or $l(v) = \mathrm{p}$ and

$$\left|\frac{\ell}{n-2} - \frac{\ell-\ell_v}{n-\mathrm{level}(v)-2}\right| \le \frac{1}{n^{1/6}}.$$

Otherwise, $v$ is called *bad*.

Using (6), we calculate:

$$\begin{aligned}
\mathrm{E}(S_2^T + L_2^T \mid s, \ell) &= \sum_{v \in T, l(v)=\mathrm{q}} p_{s,\ell}^v \cdot \frac{s - s_v}{n-2-\mathrm{level}(v)} + \sum_{v \in T, l(v)=\mathrm{p}} p_{s,\ell}^v \cdot \frac{\ell - \ell_v}{n-2-\mathrm{level}(v)} \\
&= \sum_{v \in T, l(v)=\mathrm{q}} p_{s,\ell}^v \cdot \frac{s}{n-2} + \sum_{v \in T, l(v)=\mathrm{p}} p_{s,\ell}^v \cdot \frac{\ell}{n-2} + \\
&\qquad \sum_{v \in T, l(v)=\mathrm{q}} p_{s,\ell}^v \left( \frac{s - s_v}{n-2-\mathrm{level}(v)} - \frac{s}{n-2} \right) + \\
&\qquad \sum_{v \in T, l(v)=\mathrm{p}} p_{s,\ell}^v \left( \frac{\ell - \ell_v}{n-2-\mathrm{level}(v)} - \frac{\ell}{n-2} \right) \\
&\le \sum_{v \in T, l(v)=\mathrm{q}} p_{s,\ell}^v \cdot \frac{s}{n-2} + \sum_{v \in T, l(v)=\mathrm{p}} p_{s,\ell}^v \cdot \frac{\ell}{n-2} + \\
&\qquad \sum_{\substack{v \in T, l(v)=\mathrm{q} \\ v \text{ good}}} p_{s,\ell}^v \left( \frac{1}{n^{1/6}} \right) + \sum_{\substack{v \in T, l(v)=\mathrm{q} \\ v \text{ bad}}} p_{s,\ell}^v + \\
&\qquad \sum_{\substack{v \in T, l(v)=\mathrm{p} \\ v \text{ good}}} p_{s,\ell}^v \left( \frac{1}{n^{1/6}} \right) + \sum_{\substack{v \in T, l(v)=\mathrm{p} \\ v \text{ bad}}} p_{s,\ell}^v \\
&= \frac{s}{n-2} \cdot f_{s,\ell}^{\mathrm{q}} + \frac{\ell}{n-2} \cdot f_{s,\ell}^{\mathrm{p}} + o(n) + \sum_{v \in T, v \text{ bad}} p_{s,\ell}^v \\
&= \frac{s}{n-2} \cdot f_{s,\ell}^{\mathrm{q}} + \frac{\ell}{n-2} \cdot f_{s,\ell}^{\mathrm{p}} + o(n) + \\
&\qquad \sum_{i=0}^{n-3} \Pr(\text{a bad node on level } i \text{ is reached}), \qquad (8)
\end{aligned}$$

where the first and second summand follow by the definition of $f_{s,\ell}^{\mathrm{p}}$ and $f_{s,\ell}^{\mathrm{q}}$. For the third summand, consider each of the levels of the decision tree separately. Since the probabilities $p_{s,\ell}^v$ for nodes $v$ on the same level sum up to 1, the contribution of the $1/n^{1/6}$ terms is bounded by $o(n)$. Note that this calculation yields an upper bound. However, a similar calculation shows that $s/(n-2) \cdot f_{s,\ell}^{\mathrm{q}} + \ell/(n-2) \cdot f_{s,\ell}^{\mathrm{p}} + o(n)$ is a lower bound for $\mathrm{E}(S_2^T + L_2^T \mid s, \ell)$ as well. So, to show (7) it remains to bound the last summand in (8) by $o(n)$.

To see this, consider a random input that is classified using $T$. We will show that with very high probability we do not reach a bad node in the decision tree for the first $n - n^{3/4}$ levels. 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 $s_i = X_1 + \ldots + X_i$ and $\ell_i = Y_1 + \ldots + Y_i$. We will use the method of averaged bounded differences to show that these random variables are tightly concentrated around their expectation.

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

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

*Proof.* We prove the first inequality.

First, we calculate the difference $c_j$ between the expectation of $s_i$ conditioned on $X_1, \ldots, X_j$ resp. $X_1, \ldots, X_{j-1}$ for $1 \le j \le i$.

Using linearity of expectation we may calculate

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

In this situation we may apply the following bound known as the method of averaged bounded differences (see [5, Theorem 5.3]):

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

and get

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

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

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

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

That means that for each of the first $i \le n - n^{3/4}$ levels, we are with very high probability in a *good node* on level $i$, because the deviation from the ideal case that we see a small element with probability $s/(n-2)$ is at most $1/n^{1/12}$. Thus, for the first $n - n^{3/4}$ levels the contribution of the sums of the probabilities of bad nodes is not more than $o(n)$ to the last summand in (8). For the last $n^{3/4}$ levels of the tree, we use that the contribution of the probabilities that we reach a bad node on level $i$ is at most 1 for a fixed level.

This shows that the last summand in (8) is $o(n)$. Substituting this result in (5) proves the lemma. $\square$

There are two technical complications when using this lemma in analyzing a strategy that is turned into a dual pivot quicksort algorithm. The cost bound is $a \cdot n + o(n)$. Equation (4) cannot be applied directly to such partitioning costs. Furthermore, the $o(n)$ term in Lemma 1 will get out of control for subarrays appearing in the recursion that are too small. However, the next theorem says that the leading term of (4) applies to this situation as well, although we get an error term of $o(n \ln n)$ instead of $O(n)$.

**Theorem 1.** *Let $\mathcal{A}$ be a dual pivot quicksort algorithm that gives rise to a decision tree $T_n$ for each subarray of length $n$. Assume $\mathrm{E}(P_n^{T_n}) = a \cdot n + o(n)$ for all $n$, for some constant $a$. Then $\mathrm{E}\left(C_n^{\mathcal{A}}\right) = \frac{6}{5} an \ln n + o(n \ln n)$.*

*Proof.* Fix an arbitrarily small number $\varepsilon > 0$. Then there is some $n_\varepsilon$ such that for all $n' \geq n_\varepsilon$ the average partitioning cost on a subarray of size $n'$ is smaller than $(a + \varepsilon)n'$. We only consider $n$ so large that $n^{1/\ln \ln n} \geq n_\varepsilon$ and that $(\ln n)/\ln \ln n < \varepsilon \ln n$. We split the analysis of the average cost into two parts. For each subarray of length $n' \geq n_0 = n^{1/\ln \ln n}$ the average partitioning cost is at most $(a + \varepsilon)n'$; for each subarray of size $n' < n_0$ we charge $(a + \varepsilon)n'$ to the first part of the analysis. Then (4) can be used to estimate the contribution of the first part as $\frac{6}{5}(a + \varepsilon)n \ln n + O(n) = \frac{6}{5} an \ln n + \frac{6}{5} \varepsilon n \ln n + O(n)$. In the second part we collect the contributions from splitting subarrays of size $n' < n_0$ not captured by the first part. Each such contribution is bounded from above by $2n'$ (even in absolute value), and from below by $n'$. The second part of the analysis consists in adding $2n'$ resp. $n'$ for each subarray of size $n' < n_0$ and 0 for each larger subarray. We will focus on the upper bound. Calculations for the lower bound are analogous. To this end, we wait until the algorithm has created a subarray of size $n' < n_0$ and assess the total contribution from the recursion starting from this subarray as not more than $\frac{12}{5} n' \ln n' + O(n')$, by (4). This means we must sum $\frac{12}{5} n_i \ln n_i + O(n_i)$, $1 \leq i \leq k$, over a sequence of disjoint subarrays of length $n_1, \ldots, n_k$. Since all $n_i$ are smaller than $n_0$, $n_1 + \ldots + n_k \leq n$, and since $x \mapsto x \ln x$ is a convex function, this sums up to no more than $\frac{n}{n_0} \cdot \frac{6}{5} \cdot 2n_0 \ln n_0 + O(n) = \frac{12}{5} n \ln(n_0) + O(n) < \varepsilon n \ln n + O(n)$ for $n$ large enough. Adding both parts, and choosing $n$ so large that the two $O(n)$ terms can be bounded by $\frac{4}{5} \varepsilon n \ln n$ we get the bound $\frac{6}{5} an \ln n + 3\varepsilon n \ln n$, which is sufficient since $\varepsilon$ was arbitrary. $\square$

Lemma 1 and Theorem 1 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 (w.r.t. the average comparison count), we should try to make $f^{\mathrm{q}}_{s,\ell} \cdot s + f^{\mathrm{p}}_{s,\ell} \cdot \ell$ small for each pair $s, \ell$.

## 4 Analysis of Some Known Classification Strategies

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

*Oblivious Strategies.* We will first consider strategies that do not use information of previous classifications for future classifications. To this end, we call a decision tree *oblivious* if for each level all nodes $v$ on this level share the same label $l(v)$, i. e., either in all nodes on a given level an element is compared with the smaller pivot first, or in all nodes on a given level an element is compared with the larger pivot first. 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 $f_n^{\mathrm{q}}$ denote the average number of comparisons to the larger pivot first. By assumption this value is independent of $s$ and $\ell$. Hence these strategies make sure that $f_{s,\ell}^{\mathrm{q}} = f_n^{\mathrm{q}}$ and $f_{s,\ell}^{\mathrm{p}} = n - 2 - f_n^{\mathrm{q}}$ for all pairs of values $s, \ell$.

Applying Lemma 1 gives us

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

Using Theorem 1 we get $\mathrm{E}(C_n) = 2n\ln n + o(n\ln n)$—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 [15, Section 3.2], Yaroslavskiy's algorithm is an implementation of the following strategy $\mathcal{Y}$: *Compare $\ell$ elements to $q$ first, and compare the other elements to $p$ first.* We get that $f_{s,\ell}^{\mathrm{q}} = \ell$ and $f_{s,\ell}^{\mathrm{p}} = s + m$. Applying Lemma 1, we calculate $\mathrm{E}(P_n) = \frac{19}{12}n + o(n)$. Using Theorem 1 gives $\mathrm{E}(C_n) = 1.9n\ln n + o(n\ln n)$, as in [15].

*Sedgewick's Algorithm.* Following [15, 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 $f^{\mathrm{p}}_{s,\ell} = (n-2)\cdot \ell/(s+\ell)$. Using Lemma 1, we calculate $\mathrm{E}(P_n) = \frac{16}{9}n+o(n)$. Applying Theorem 1 gives $\mathrm{E}(C_n) = 2.133... \cdot n \ln n + o(n \ln n)$, as known from [15].

Obviously, this is worse than strategy $\mathcal{P}$ considered before.[3] 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 above, we get $\mathrm{E}(P_n) = \frac{14}{9}n + o(n)$ which yields $\mathrm{E}(C_n) = 1.866... \cdot n \ln n + o(n \ln n)$—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 [14].

*Remark.* Exchanging $f^{\mathrm{p}}_{s,\ell}$ and $f^{\mathrm{q}}_{s,\ell}$ as in the strategy described above is a general technique. In fact, if the leading coefficient of the average number of comparisons for a fixed choice of $f^{\mathrm{p}}_{s,\ell}$ and $f^{\mathrm{q}}_{s,\ell}$ is $\alpha$, e. g., $\alpha = 2.133...$ for strategy $\mathcal{S}$, then the leading coefficient of the strategy that exchanges $f^{\mathrm{p}}_{s,\ell}$ and $f^{\mathrm{q}}_{s,\ell}$ 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 all other elements to $q$ first. In this section, we will study the following strategy $\mathcal{N}$: *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 all comparisons have been made. We shall analyze a guessing strategy to resolve this.

### 5.1 Analysis of the Ideal 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 $\ell$ this means that for $s > \ell$

[3] We remark that in his thesis Sedgewick [10] focused on the average number of swaps, not on the comparison count.

the classification strategy makes exactly $\ell$ additional comparisons, and for $s \leq \ell$ it makes $s$ additional comparisons.

Since we can exactly describe the average cost term, we directly apply (5) and do not need Theorem 1. A standard calculation shows that

$$\mathrm{E}(P_n) = \frac{4}{3}n + \frac{1}{\binom{n}{2}} \sum_{s+\ell \leq n-2} \min(s,\ell) = \frac{3}{2}n + O(1). \tag{9}$$

Applying (4), we get $\mathrm{E}(C_n) = 1.8 n \ln n + O(n)$, which is by $0.1 n \ln n$ 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 1 the average comparison count is determined up to lower order terms by the parameters $f^{\mathrm{q}}_{s,\ell}$ and $f^{\mathrm{p}}_{s,\ell} = n - 2 - f^{\mathrm{q}}_{s,\ell}$. Strategy $\mathcal{N}$ chooses these values such that $f^{\mathrm{q}}_{s,\ell}$ is either 0 or $n-2$, minimizing each term of the sum in Lemma 1—and thus minimizing the sum.

## 5.2 Guessing Whether $s < \ell$ or not

We explain how the ideal classification strategy just described can be approximated by an implementation. The idea simply 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 $sz = o(n)$ many elements (the *sample*) and calculate $s'$ and $\ell'$, the number of small and large elements in the sample. If $s' < \ell'$, compare the remaining $n - 2 - sz$ elements with $q$ first, otherwise compare them with $p$ first. We say that the guess was *correct* if the relation "$s' < \ell'$" correctly reflects whether $s < \ell$ or not.

We incorporate guessing errors into (9) as follows:

$$\begin{aligned}\mathrm{E}(P_n) &= \frac{4}{3}n + \frac{1}{\binom{n}{2}} \sum_{s+\ell \leq n-2} \Big( \Pr(\text{guess correct}) \cdot \min(s,\ell) + \\ &\qquad\qquad \Pr(\text{guess wrong}) \cdot \max(s,\ell) \Big) + o(n) \\ &= \frac{4}{3}n + \frac{2}{\binom{n}{2}} \sum_{s=0}^{n/2} \sum_{\ell=s+1}^{n-s} \Big( \Pr(\text{guess correct}) \cdot s + \\ &\qquad\qquad \Pr(\text{guess wrong}) \cdot \ell \Big) + o(n). \end{aligned} \tag{10}$$

The following lemma says that for a wide range of values $s$ and $\ell$ the probability of a guessing error is exponentially small.

**Lemma 2.** *Let $s$ and $\ell$ with $s \leq \ell - n^{3/4}$ and $\ell \geq n^{3/4}$ for $n \in \mathbb{N}$ be given. Let $sz = n^{2/3}$. Then $\Pr(s' > \ell') \leq \exp\left(-2n^{1/6}/9\right)$.*

*Proof.* Let $sz = n^{2/3}$. Let $X_i$ be a random variable that is $-1$ if the $i$-th classified element is large, 0 if it is medium, and 1 if it is small. Let $d = \sum_{i=1}^{sz} X_i$.

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

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

To see this, we let $s_i$ resp. $\ell_i$ denote the number of small resp. large elements that are still present if $X_1, \ldots, X_i$ are classified. Furthermore, let $Y_i$ be the 0-1 random variable that is 1 iff $X_i$ is 1, and let $Z_i$ be the 0-1 random variable that is 1 iff $X_i$ is $-1$.

We calculate:

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

The method of averaged bounded differences (see [5, Theorem 5.3]) says that

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

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

$$\Pr(d > 0) \le \exp\left( -\frac{2n^{1/6}}{9} \right). \qquad \square$$

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

Our classification strategy will now work as follows. It starts by classifying $sz = n^{2/3}$ many elements and then decides which pivot is used in the first comparison for the remaining $n - sz$ elements. Then it classifies the elements according to this decision.

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

**Theorem 2.** *Let $sz = n^{2/3}$. Then the average comparison count of the strategy described above turned into a dual pivot quicksort algorithm is $1.8n \ln n + o(n \ln n)$.*

*Proof.* First, classify $sz$ many elements. The number of key comparisons for these classifications is at most $2n^{2/3} = o(n)$. By symmetry, we may focus on the case that $s \leq \ell$. We distinguish the following three cases:

1. $\ell \leq n^{2/3}$: The contribution of terms in (10) for this case is at most

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

2. $\ell - n^{2/3} \leq s \leq \ell$: The contribution of terms in (10) in this case is at most

$$\frac{2}{\binom{n}{2}} \sum_{s=0}^{n/2} \sum_{\ell=s}^{\min(s+n^{2/3}, n-s)} \ell = o(n).$$

3. $\ell \geq n^{2/3}$ and $s \leq \ell - n^{2/3}$. Let $\mathrm{m}(\ell) = \min(n-\ell, \ell - n^{2/3})$. Following Lemma 2, the probability of guessing wrong is at most $\exp(-2n^{1/6}/9)$. The contribution of this case in (10) is hence at most

$$\frac{2}{\binom{n}{2}} \sum_{\ell=n^{2/3}}^{n} \sum_{s=0}^{\mathrm{m}(\ell)} \left( s + \exp(-2n^{1/6}/9)\ell \right) = \left( \frac{2}{\binom{n}{2}} \sum_{\ell=n^{2/3}}^{n} \sum_{s=0}^{\mathrm{m}(\ell)} s \right) + o(1).$$

Thus, the contribution of sampling and estimation errors is $o(n)$. As discussed in the proof of Theorem 1, the probability bounds in Lemma 2 do not hold for small subarrays created in the recursion. Reasoning exactly as in the proof of that theorem, we get that sorting these small subarrays results in an additional summand of $o(n \ln n)$ to the average number of key comparisons.

In conclusion, we expect a partitioning step to make $\frac{3}{2}n + o(n)$ key comparisons, see (9). Applying Theorem 1, we get $\mathrm{E}(C_n) = 1.8n \ln n + o(n \ln n)$. □

While being optimal, this strategy has an error term of $o(n \ln n)$. In the next section we will present a different strategy that will be optimal up to an $O(n)$ error term.

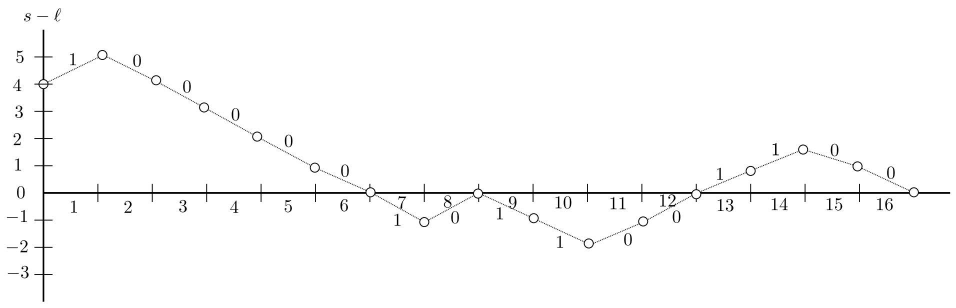

**Fig. 3.** Visualization of an example run of strategy $\mathcal{O}$ on 16 elements that are either small or large. The $y$ axis shows the difference of small and large elements that are yet unclassified, e. g., after the first element is classified it goes up by 1 because the first element in this example is large. If the strategy makes a correct choice for the first pivot, it costs 0, otherwise it costs 1. We always charge 1 if the strategy has the same number of small and large elements in the unclassified part (7th, 9th, and 13th element). So, the cost is 1 if the trajectory moves away from the time axis, and 0 if it moves towards the time axis.

## 6 An Optimal Classification Strategy With an $O(n)$ Error Term

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 $\ell$, i. e., the number of small resp. large elements, are known to the algorithm after the pivots are chosen, and the decision tree can have different node labels for each such pair of values. Recall that $s_v$ and $\ell_v$, resp., denote the number of elements that have been classified as small and large, resp., when at node $v$ in the decision tree. We consider the following strategy $\mathcal{O}$: *Given $s$ and $\ell$, 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.*[4]

**Theorem 3.** *Strategy $\mathcal{O}$ is optimal, i. e., its ACT is at most as large as $\text{ACT}_T$ for every single tree $T$. When using $\mathcal{O}$ in a dual pivot quicksort algorithm, we get* $\text{E}(C_n^{\mathcal{O}}) = 1.8n \ln n + O(n)$.

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

Now we prove the second statement. To this end, we first derive an upper bound for the number of additional comparisons by pessimistically assuming that

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

strategy $\mathcal{O}$ always makes the wrong decision when $s - s_v = \ell - \ell_v$, i. e., the number of small elements left in the input equals the number of large elements left in the input. Assume $s > \ell$ and omit medium elements in the input. Let $n'$ denote the number of small and large elements in the remaining input. Now consider the example run of strategy $\mathcal{O}$ depicted in Figure 3. In a trajectory that starts at $s - \ell \geq 0$ one can pair off the 0-edges and 1-edges (from a 1-edge go horizontally to the right until the next 0-edge at the same level is hit), excepting for the $s - \ell$ 0-edges in the first part of the trajectory that go down to a level that has not been touched before until the $x$ axis is hit. Hence the number of additional comparisons, i. e., the number of 1-edges, is $\frac{1}{2}(n' - (s - \ell)) = \ell$. If $s - \ell < 0$, then the same argument shows that the number of additional comparisons is $s$.

The additional cost term of strategy $\mathcal{O}$ (see (5)) is thus at most

$$\frac{2}{\binom{n}{2}} \cdot \sum_{\substack{s+\ell \leq n-2 \\ \ell \leq s}} \ell, \tag{11}$$

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

Now it remains to show that this is tight. For this it suffices to estimate (for a random input) the average number of positions $i, i \in \{1, \ldots, n-2\}$, in which the number of small elements in the rest of the input, i. e., at positions $i, \ldots, n-2$, equals the number of large elements in the rest of the input.

A somewhat involved calculation shows that this number is $O(\log n)$. Additionally, one can also show that a (negative) term of $O(\log n)$ decreases the total average number of key comparisons only by $O(n)$ when we use such a strategy to implement a dual pivot quicksort algorithm. The details of these calculations can be found in Appendix A. □

While being optimal w.r.t. minimizing the ACT, 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 $\mathcal{L}$: *The comparison at node $v$ is with the smaller pivot first if $s_v > l_v$, otherwise it is with the larger pivot first.*

While $\mathcal{O}$ looks into the future ("Are there more small elements or more large elements left?"), strategy $\mathcal{L}$ looks into the past ("Have I seen more small or more large elements so far?"). It is not hard to see that for a given input the number of additional comparisons of strategy $\mathcal{O}$ and $\mathcal{L}$ can differ significantly. The next theorem shows that averaged over all possible inputs, however, there is only a small difference.

**Theorem 4.** *Let $ACT_{\mathcal{O}}$ resp. $ACT_{\mathcal{L}}$ be the ACT for classifying $n$ elements using strategy $\mathcal{O}$ resp. $\mathcal{L}$. Then $ACT_{\mathcal{L}} = ACT_{\mathcal{O}} + O(\log n)$. When using $\mathcal{L}$ in a dual pivot quicksort algorithm, we get $\mathrm{E}(C_n^{\mathcal{L}}) = 1.8n \ln n + O(n)$.*

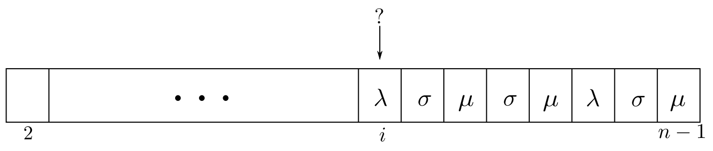
?
2
$i$
$n-1$
$\lambda$ $\sigma$ $\mu$ $\sigma$ $\mu$ $\lambda$ $\sigma$ $\mu$

**Fig. 4.** Visualization of the decision process when inspecting element $a_i$. Applying strategy $\mathcal{O}$ from left to right uses that of the remaining elements three are small and two are large, so it decides that $a_i$ should be compared with $p$ first. Applying strategy $\mathcal{L}$ from right to left uses that of the inspected elements three were small but only one was large, so it decides to compare $a_i$ with $p$ first, too. Note that the strategies would differ if one small element would be a medium element.

*Proof.* Assume that strategy $\mathcal{O}$ inspects the elements in the order $a_{n-1}, \ldots, a_2$, while $\mathcal{L}$ 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 4. The same calculation as in Appendix A shows that $\text{ACT}_{\mathcal{L}} - \text{ACT}_{\mathcal{O}}$ is $O(\log n)$, which sums up to a total additive contribution of $O(n)$ when using strategy $\mathcal{L}$ in a dual pivot quicksort algorithm. See Appendix A for details. □

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

We have now identified three optimal strategies, two of which can be used in an actual algorithm. Note that the bound for the average comparison count of strategy $\mathcal{L}$ has an additional summand of $O(n)$, while the bound for strategy $\mathcal{N}$ has an additional summand of $o(n \ln n)$. The difference in the comparison count is clearly visible in an actual implementation, as we shall see in Section 9. However, in comparison to strategy $\mathcal{N}$ strategy $\mathcal{L}$ requires an additional counter and a conditional jump for each element.

## 7 Choosing Pivots From a Sample

We consider here the variation of dual pivot quicksort in which the two pivots are chosen from a sample of 5 elements. We choose the second-largest and fourth-largest as pivots. (This is the pivot choice that is used in Yaroslavskiy's algorithm in the JRE7 implementation, see [18] for further discussion.) The probability that $p$ and $q$, $p < q$, are chosen as pivots is exactly $(s \cdot m \cdot \ell)/\binom{n}{5}$. Following Hennequin [6, pp. 52–53], for partitioning costs $\text{E}(P_n) = a \cdot n + O(1)$ we get

$$\text{E}(C_n) = \frac{1}{H_6 - H_2} \cdot a \cdot n \ln n + O(n) = \frac{20}{19} \cdot a \cdot n \ln n + O(n), \qquad (12)$$

where $H_n$ denotes the $n$-th harmonic number. Note that only strategy $\mathcal{O}$ and $\mathcal{L}$ guarantee such partitioning cost in our setting. When applying Lemma 1, we have average partitioning cost of $a \cdot n + o(n)$. Using the same argument as in the proof of Theorem 1, the average comparison count becomes $20/19 \cdot a \cdot n \ln n + o(n \ln n)$ in this case.

We will now investigate the effect on the average number of key comparisons in Yaroslavskiy's resp. the optimal partitioning method $\mathcal{N}$. The average number of medium elements remains $(n-2)/3$. For strategy $\mathcal{Y}$, we calculate

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

Applying (12), we get $\mathrm{E}(C_n^{\mathcal{Y}}) = 1.704 n \ln n + o(n \ln n)$ key comparisons. (Note that Wild *et al.* [18] calculated this leading coefficient as well.) This 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 [7]. For our proposed partitioning method from Section 5, we get

$$\mathrm{E}(P_n^{\mathcal{N}}) = \frac{4}{3}n + \frac{2}{\binom{n}{5}} \sum_{\substack{s+\ell \leq n-3 \\ s \leq \ell}} s \cdot s \cdot m \cdot \ell + o(n) = \frac{37}{24}n + o(n).$$

Again using (12), we obtain $\mathrm{E}(C_n^{\mathcal{N}}) = 1.623 n \ln n + o(n \ln n)$, which is optimal as well. By applying random sampling as above, we get an algorithm that makes $1.623 n \ln n + o(n \ln n)$ key comparisons on average, improving further on the leading coefficient compared to clever quicksort and Yaroslavskiy's algorithm.

## 7.1 Pivot Sampling in Classical Quicksort and Dual Pivot Quicksort

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

The following proposition, which is a special case of [6, Proposition III.9 and Proposition III.10], will help in this discussion.

**Proposition 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*

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

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

**Table 1.** Comparison of the leading term of the average cost of classical quicksort and dual pivot quicksort for specific sample sizes. Note that for real-world input sizes, however, the linear term can make a big difference.

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

For an optimal dual pivot partitioning algorithm, we get the following: 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 can be calculated as follows:

$$\mathrm{E}(P_n^k) = \frac{4}{3}n + O(1) + \frac{2}{\binom{n}{k}} \sum_{\substack{s+\ell\leq n-2\\ s\leq \ell}} \binom{s}{(k-2)/3}\binom{m}{(k-2)/3}\binom{\ell}{(k-2)/3}\cdot s. \tag{13}$$

Unfortunately, we could not find a closed form of $\mathrm{E}(P_n^k)$. Some calculated values in which classical and dual pivot quicksort use the same sample size can be found in Table 1. These values clearly indicate that starting from a sample of size 5 classical quicksort has a smaller average comparison count than dual pivot quicksort. Furthermore, while it is well known that for classical quicksort in which the pivot is chosen as the median of a sample, the average comparison count converges with increasing sample size to lower bound of $(1/\ln(2))\cdot n\ln n + O(n)$ comparisons on average for comparison based sorting, this does not seem to be the case for dual pivot quicksort. It is an interesting question which multi-pivot quicksort algorithms converge to the optimal value and which do not. We strongly believe that the (obvious) symmetric partitioning algorithms using $2^k-1$ pivots (experimentally studied by Tan [13]) are all optimal for sample sizes of, e. g., $\sqrt{n}$.

## 8 Swaps In Dual Pivot Quicksort

After the discussion of key comparisons in dual pivot quicksort, we will now turn the focus on minimizing swap operations. Fortunately, we can build on work that has been done for the so-called *Dutch National Flag* problem. This problem is defined as follows: Given an array containing $n$ elements, where each element is either red, blue or white, rearrange the elements using swaps such that they resemble the national flag of the Netherlands (red followed by white followed by blue). One immediately sees the connection between this problem and the partitioning problem in the context of dual pivot quicksort, for the partitioning problem deals with classifying elements into three different types called small, medium, and large, resp, and rearranging these elements such that small elements are followed by medium elements which are followed by large elements. We quickly review the work done on this problem, and then analyze the average swap count of these algorithms when using them for dual pivot quicksort. Many results on the DNF cannot be applied directly, since the probability spaces differ.

## 8.1 Algorithms for the Dutch National Flag Problem

Here, we describe three algorithms that solve the DNF problem. We state them in our notation, using small, medium, and large elements, resp., instead of red, white, and blue elements, resp. We assume that $A$ is an array of length $n$ containing small, medium, and large elements.

The first algorithm is due to Dijkstra [4].

**Algorithm 1 (SwapA).**

```
procedure DijsktraSwap($A[1..n]$)
1     i := 1; j := $n$; k := $n$
2     while i ≤ j
3         classify $A$[j]
4             case small: swap(i,j); i++;
5             case medium: j−−;
6             case large: swap(j,k); j−−; k−−;
7     end while
```

The second algorithm is due to Meyer [9]. It avoids swapping elements that have not been inspected so far (that might be small and hence already present on a suitable position) as done in Algorithm 1 on line 4.

**Algorithm 2 (SwapB).**

```
procedure MeyerSwap($A[1..n]$)
1     i := 1; j := $n$; k := $n$
2     while i ≤ j
3         classify $A$[j]
4             case small:
5                 while $A$[i] is small
6                     i++;
7                 end while
8                 if i < j
9                     swap(i,j); i++;
10            case medium: j−−;
11            case large: swap(j,k); j−−; k−−;
```

```
    end while
```

The next algorithm was explicitly stated by Chen in [2], but has been discussed in [11], too. Instead of classifying elements and moving them to their final position, it uses two stages. In the first stage, it moves all small elements to the left of the array. The second stage moves all large elements to the right, but does not inspect the array part that contains the small elements.

**Algorithm 3 (SwapC).**

```
procedure SwapC($A[1..n]$)
    i := 1; j := $n$;
    while i < j
        while $A$[i] is small and i < j
            i++;
        end while
        while $A$[j] is not small and i < j
            j−−;
        end while
        if i < j
            swap(i,j); i++; j−−;
    end while
    if i < $n$
        j := $i$; k := $n$;
        while j < k
            while $A$[j] is not large and j < k
                j++;
            end while
            while $A$[k] is large and j < k
                k−−;
            end while
            if j < k
                swap(j,k); j++; k−−;
    end while
```

### 8.2 Analysis of the Average Swap Count

Given a strategy, e. g., the optimal strategy from Section 5, we get a partitioning procedure for dual pivot quicksort by using Algorithm 1 or Algorithm 2—replacing all classification steps according to the given strategy. In fact, Yaroslavskiy's algorithm uses strategy $\mathcal{Y}$ together with Meyer's algorithm.

The situation is a little bit different with Algorithm 3, for it uses that the smaller pivot is used for the first comparison. (This case was analyzed as strategy $\mathcal{P}$ in Section 4.)

For now, we focus on calculating the average swap count of these algorithms. Note that these algorithms have been analyzed in the setting of the Dutch national flag problem (see, e. g., [2]), but in a different probability space, in which each element independently chooses to be small, medium or large with probability 1/3.

*Analysis of Algorithm 1.* Algorithm 1 has the following properties with respect to swap operations:

(i) Each small element causes exactly one swap.
(ii) Each large element causes exactly one swap.
(iii) No medium element causes a swap operation.

For the average swap count $P^1_{S,n}$ for partitioning an array of length $n$—conditioning on the pivot choices as before—, we get the following formula:[5]

$$\mathrm{E}(P^1_{S,n}) = \frac{1}{\binom{n}{2}} \sum_{s+\ell\leq n} (s+\ell) = \frac{2}{3}n + O(1).$$

Note that (4) also holds for the swap case (*cf.* [15]). For the total average number of swaps $S_n$ for sorting an array of length $n$ using a dual pivot quicksort approach, we hence obtain $\mathrm{E}(S^1_n) = 0.8n \ln n + O(n)$.

*Analysis of Algorithm 2.* Algorithm 2 has the following properties with respect to swap operations for $s$ small and $\ell$ large elements:

(i) Each small element that resides in the array positions $s+1,\ldots,n$ causes exactly one swap.
(ii) Each large element causes exactly one swap.
(iii) No medium element causes a swap.

For the average swap count $P^2_{S,n}$ for Algorithm 2, we get

$$\mathrm{E}(P^2_{S,n}) = \frac{1}{\binom{n}{2}} \sum_{s+\ell\leq n} (s\cdot(n-s)/n+\ell) = \frac{1}{2}n + O(1),$$

which yields a total average swap count of $\mathrm{E}(S^2_n) = 0.6n \ln n + O(n)$. To no surprise, this is exactly the same number as the average swap count calculated by Wild and Nebel [15] of Yaroslavskiy's algorithm.

When we use the optimal strategy of Section 5, we can further improve this algorithm as follows: After the random sampling step, we make a guess whether $s > \ell$. If this is the case, we use Algorithm 2 and always compare to the smaller pivot first. Otherwise, we slightly change Algorithm 2 by letting `j` run from 1 to `k`, swapping the small and large case, i. e., using the inner while loop when $A$[`j`] is large to ignore large elements. Conditioning on $s$ and $\ell$, almost the same calculations as above show that this algorithm yields a total average swap count of $0.45n \ln n + O(n)$.

[5] When analyzing a partitioning method for dual pivot quicksort, we have to consider that pivots have to be placed into their correct position at the end. In total, we get an additive summand of 2 to the average swap count. We omit this summand here.

*Analysis of Algorithm 3.* Algorithm 3 has the following behavior with respect to swap operations for $s$ small and $\ell$ large elements.

(i) In the first stage, each small element that resides in the array positions $s+1, \ldots, n$ causes exactly one swap.
(ii) In the second stage, each large element that resides in the array positions $s+1, \ldots, n-l-1$ causes exactly one swap.
(iii) No other element causes a swap.

Given a random array containing $s$ small, $m$ medium, and $\ell$ large elements, note that after the first stage the sequence of array elements in positions $s+1, \ldots, n$ is still random, i. e., each such sequence is equally likely. (This is the same as the property that subarrays occurring in recursive steps of quicksort are fully random, *cf.* [15, Section 3.1].)

For the average swap count $P^3_{S,n}$ for Algorithm 3, we get

$$\mathrm{E}(P^3_{S,n}) = \frac{1}{\binom{n}{2}} \sum_{s+\ell \leq n} (s \cdot (n-s)/n + \ell \cdot m/(n-s)) = \frac{5}{18}n + O(1),$$

which gives $\mathrm{E}(S^3_n) = 0.33..n \ln n + O(n)$—and hence significantly decreases the number of swap operations compared to the previous algorithms.

Again, we can use the optimal strategy of Section 5 by deciding if we use Algorithm 3 or its modified version, in which we first split into large and non-large elements after the random sampling step.

### 8.3 A Simple Lower Bound on the Average Swap Count

While Chen's algorithm decreases the average swap count compared to Meyer's algorithm, it is not clear whether it achieves the lowest possible average swap count. In the related DNF problem, there exists an algorithm [1] that is asymptotically optimal. However, this algorithm heavily relies on the (different!) randomness assumptions in this particular problem. We give a simple lower bound for our setting.

To achieve this bound, we simply calculate the average number of misplaced elements in the input, i. e., medium and large elements in the input which reside in array cells that are left of the final position of the smaller pivot, small and large elements that reside in array cells that lie in between the two pivots, and small and medium elements that lie right of the larger pivot. Actually, these exact numbers have been calculated by Wild and Nebel, see [15, Table 3]. They show that each "group of misplaced elements", e. g., small elements in the input that reside in a location right of the larger pivot, has exactly $(n-3)/12$ elements on average. Adding together all these values gives an average number of about $n/2$ elements. Now, with a single swap operation we can move at most 2 elements into their final position. Thus, at least $n/4$ swap operations are needed to move these misplaced elements to their final locations. This is smaller than the $5/18n + O(1)$ swap operation Chen's algorithm needs on average.

It is an interesting open question whether Chen's algorithm is optimal or not (as in the DNF problem) w.r.t. minimizing the average swap count.

## 9 Experiments

We have implemented the methods presented in this paper in C++. These algorithms have not been fine-tuned; this section is hence meant to provide only preliminary results and does not replace a thorough experimental study. Our experiments were carried out on an Intel Xeon E5645 at 2.4 GHz with 48 GB Ram running Ubuntu 12.04 with kernel version 3.2.0. For measuring running times, we used three different scenarios.

1. C++ code compiled with *gcc* using no optimization flags,
2. C++ code compiled with *gcc* using the *-O2* optimization flag, and
3. Java code using Oracle's Java 7.

We have incorporated random sampling into the partitioning step of our method (strategy $\mathcal{N}$) by comparing the first $n' = \max(n/100, 7)$ elements with $p$ first. We switched to comparing with $q$ first if the algorithm has seen more large than small elements after $n'$ steps.

In Section 9.1, we give an experimental evaluation of the comparison and swap count of the algorithms considered in this paper. In Section 9.2, we focus on the actual running times needed to sort a given input. The charts of our experiments can be found at the end of this paper.

### 9.1 Comparison and Swap Count

We first have a look at the comparison and swap count needed to sort a random input of up to 50 000 000 integers. We did not switch to a different sorting algorithm, e. g., insertion sort, to sort short subarrays.

Figure 6 shows the results of our experiments for algorithms that choose the pivots directly. We see that the linear term in the average comparison count has a big influence on the number of comparisons. For Yaroslavskiy's algorithm this linear term is $-2.46n$, as calculated by the authors of [15].

The results confirm our theoretical studies and show that our algorithm beats all other algorithms with respect to the comparison count, although we incorporated random sampling directly into the partitioning step. We also see that the modified version of Sedgewick's algorithm beats Yaroslavskiy's algorithm. On the other hand, Sedgewick's original algorithm makes the most key comparisons and is even worse than standard quicksort, as described in Section 4.

Figure 7 shows the same experiment for the algorithms that choose the pivots from a small sample. This plot confirms the theoretical results from Section 7.

Figure 8 shows the same experiment for the swap count. These results confirm the theoretical study conducted in Section 8. We see that Sedgewick's strategy and Dijkstra's strategy make the most swaps. Yaroslavskiy's algorithm is clearly better and shows exactly the behavior calculated for a strategy that uses Meyer's algorithm. We see that the modified version of Meyer's algorithm that makes a small random sampling step and then decides which version it uses is better. The best swap strategy—as calculated in Section 8—is Algorithm 3. It almost matches the swap count of standard quicksort. However, in our initial experiments, the running time of this variant was not competitive to algorithms that used Algorithm 1 or Algorithm 2.

### 9.2 Running Times

We now consider the running times of our algorithms to sort a given input. To measure running times, all algorithms used a small sample to choose the pivots from. Clever quicksort uses the median of a sample of three elements. Yaroslavskiy and our algorithm use the second- and fourth-largest elements from a sample of five elements. We sorted subarrays of size at most 16 directly using insertion sort.

With respect to running times, we see that Yaroslavskiy's algorithm is superior to the other algorithms when sorting random permutations of $\{1, \ldots, n\}$. Our method is about 3% slower, see Figure 9. When sorting strings (and key comparisons become more expensive), our algorithm can actually beat Yaroslavskiy's algorithm, see Figure 9. However, the difference is only about 2%, see Figure 10. Note that Java 7 does not use Yaroslavskiy's algorithm when sorting strings.

## 10 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 lower order terms and even up to $O(n)$. This minimum is $1.8n \ln n + O(n)$. We pointed out the obvious relation between dual pivot quicksort and the dutch national flag problem, this identifying an algorithm which decreases the average swap count of $0.6n \ln n + O(n)$ (Yaroslavskiy's algorithm) to $0.33..n \ln n + O(n)$.

Several open questions remain. While we are now in a situation where we can compare classical and dual pivot quicksort variants w.r.t. the average comparison count resp. the average swap count, it does not seem to be the case that these cost measures can fully explain the different running time behavior observed in practice. To this end, one might look into different cost measures that explain these differences more accurately. From an engineering point of view, one might look for a clever implementation of our classification idea that beats Yaroslavskiy's algorithm. While we restricted ourselves to analyze the average comparison count and average swap count, one can of course study the actual distribution of these random variables in much more detail, as, e. g., done for Yaroslavskiy's algorithm by Wild *et al.* [17]. Moreover, it has been shown by Wild *et al.* in [16] that in difference to quicksort, the well-known quickselect algorithm for finding order statistics is not improved by using Yaroslavskiy's algorithm in the partitioning step. It is an interesting question whether this holds for all dual pivot approaches.

**Acknowledgements.** 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 Michael Rink for interesting and useful discussions. We thank the referees of the conference submission for their insightful comments, which helped a lot in improving the presentation.

## References

1. Bitner, J.R.: An asymptotically optimal algorithm for the dutch national flag problem. SIAM J. Comput. 11(2), 243–262 (1982)
2. Chen, W.M.: Probabilistic analysis of algorithms for the dutch national flag problem. Theor. Comput. Sci. 341(1-3), 398–410 (2005)
3. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms (3. ed.). MIT Press (2009)
4. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall (1976)
5. Dubhashi, D.P., Panconesi, A.: Concentration of Measure for the Analysis of Randomized Algorithms. Cambridge University Press (2009)
6. Hennequin, P.: Analyse en moyenne d'algorithmes: tri rapide et arbres de recherche. Ph.D. thesis, Ecole Politechnique, Palaiseau (1991)
7. Hoare, C.A.R.: Quicksort. Comput. J. 5(1), 10–15 (1962)
8. Knuth, D.E.: The Art of Computer Programming, Volume III: Sorting and Searching. Addison-Wesley (1973)
9. Meyer, S.J.: A failure of structured programming. Zilog Corp., Software Dept. Technical Rep. No. 5, Cupertino, CA (1978)
10. Sedgewick, R.: Quicksort. Ph.D. thesis, Standford University (1975)
11. Sedgewick, R.: Quicksort with equal keys. SIAM J. Comput. 6(2), 240–268 (1977)
12. Sedgewick, R., Flajolet, P.: An introduction to the analysis of algorithms. Addison-Wesley-Longman (1996)
13. Tan, K.H.: An asymptotic analysis of the number of comparisons in multipartition quicksort. Ph.D. thesis, Carnegie Mellon University (1993)
14. Wild, S.: Java 7's Dual Pivot Quicksort. Master's thesis, University of Kaiserslautern (2013)
15. Wild, S., Nebel, M.E.: Average case analysis of Java 7's dual pivot quicksort. In: ESA'12. pp. 825–836 (2012)
16. Wild, S., Nebel, M.E., Mahmoud, H.: Analysis of quickselect under Yaroslavskiy's dual-pivoting algorithm. CoRR abs/1306.3819 (2013)
17. Wild, S., Nebel, M.E., Neininger, R.: Average case and distributional analysis of Java 7's dual pivot quicksort. CoRR abs/1304.0988 (2013)
18. Wild, S., Nebel, M.E., Reitzig, R., Laube, U.: Engineering Java 7's dual pivot quicksort using MaLiJan. In: ALENEX'13 (2013)

## A Missing Details of the Proof of Theorem 3

In Section 6 we have analyzed the additional comparisons count of the optimal strategy $\mathcal{O}$. A simple argument showed that for fixed pivots, the number of additional comparisons on an input is at most $s$ when $s \leq \ell$, and $\ell$ otherwise. This showed that the total average sorting cost when this strategy is turned into a dual pivot algorithm is at most $1.8n \ln n + O(n)$. We got this upper bound by assuming that the optimal strategy always decides for the wrong pivot when there are the same number of small elements as the number of large elements in the unclassified part of the input. We will now analyze how much this upper bound deviates from the exact average additional cost term.

For this assume that the optimal strategy $\mathcal{O}$ inspects the elements in the order $(a_1, \ldots, a_n)$. (For simplicity of the calculation, we assume that $n$ elements are to be classified, i. e., the input consists of $n+2$ elements including the two pivots.) We have to calculate the average number of positions $i \in \{1, \ldots, n\}$ where we have the same number of small elements as large elements in $(a_i, \ldots, a_n)$. We call a position $i$ where this happens a *zero-crossing* at position $i$.

By symmetry, we may assume that the number of small elements is at most as large as the number of large elements. We temporarily omit medium elements to simplify calculations, i. e., we assume that the number of small and large elements is $n$. Let $Z_n$ be the random variable that counts the number of zero-crossings for an input of $n$ elements. We calculate:

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

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

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

We continue by bounding the product.

$$
\begin{aligned}
\mathrm{E}(Z_n) &= \Theta\left(\frac{1}{n}\right) \sum_{i=1}^{n/2} \frac{n+1}{\sqrt{i}(n-2i+1)} \sum_{j=0}^{n/2-i} \prod_{k=0}^{i-1} \left(1 - \left(\frac{2j}{n-2k}\right)^2\right) \\
&= \Theta\left(\frac{1}{n}\right) \sum_{i=1}^{n/2} \frac{n+1}{\sqrt{i}(n-2i+1)} \sum_{j=0}^{n/2-i} \prod_{k=0}^{i-1} \left(1 - \left(\frac{2j}{n}\right)^2\right) \\
&= \Theta\left(\frac{1}{n}\right) \sum_{i=1}^{n/2} \frac{n+1}{\sqrt{i}(n-2i+1)} \sum_{j=0}^{n/2-i} \left(1 - \left(\frac{2j}{n}\right)^2\right)^i \\
&= \Theta\left(\frac{1}{n}\right) \sum_{i=1}^{n/2} \frac{n+1}{\sqrt{i}(n-2i+1)} \sum_{j=0}^{n/2} \left(1 - \left(\frac{2j}{n}\right)^2\right)^i \\
&= \Theta\left(\frac{1}{n}\right) \sum_{i=1}^{n/2} \frac{n+1}{\sqrt{i}(n-2i+1)} \left(\int_0^{n/2} \left(1 - \left(\frac{2t}{n}\right)^2\right)^i \mathrm{d}t + 1\right).
\end{aligned}
$$

Using a standard computer algebra system for the integral, we obtain

$$
\mathrm{E}(Z_n) = \Theta\left(\frac{1}{n}\right) \sum_{1 \le i \le n/2} \frac{n+1}{\sqrt{i}(n-2i+1)} \cdot \left(n \cdot \frac{\Gamma(i+1)}{\Gamma(i+3/2)}\right),
$$

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

$$\begin{aligned}
\mathrm{E}(Z_n) &= \Theta(1) \cdot \sum_{1\le i\le n/2} \frac{n+1}{i(n-2i+1)} \\
&= \Theta(1) \cdot \left( \sum_{1\le i\le n/4} \frac{1}{i} + \sum_{n/4+1\le i\le n/2} \frac{1}{n-2i+1} \right) = \Theta(\log n).
\end{aligned}$$

Now we consider the case that the input contains medium elements. Fix the number of small elements and the number of large elements. Since medium elements have no influence on the value of the additional cost term, we have that $\mathrm{E}(Z_n \mid s \text{ small}, m \text{ medium}, \ell \text{ large elements}, s+m+l=n)$ equals $\mathrm{E}(Z_{s+\ell} \mid s \text{ small}, \ell \text{ large elements})$, i. e., it equals the average number of zero-crossings for a smaller input containing $s+\ell$ elements.

We can now simply calculate the average number of zero-crossing for an arbitrary input as follows:

$$\begin{aligned}
\mathrm{E}(Z_n) &= \frac{1}{\binom{n}{2}} \sum_{s+\ell\le n} \mathrm{E}(Z_n \mid s, \ell) \\
&= \frac{1}{\binom{n}{2}} \sum_{s+\ell\le n} \mathrm{E}(Z_{s+\ell} \mid s, \ell) \\
&= \frac{1}{\binom{n}{2}} \sum_{s+\ell\le n} \Theta(\log(s+\ell)) = \Theta(\log n),
\end{aligned}$$

which concludes the first part of the proof of Theorem 3.

We now know that the difference between the upper bound on the additional cost term shown in (11) and the actual additional cost term is $\Theta(\log n)$. It remains to show that the influence of these $\Theta(\log n)$ terms for the total average sorting cost is bounded by $O(n)$. By linearity of expectation, we consider these terms in the average sorting cost of (3) separately. So, assume that the cost associated with a partitioning step involving a subarray of length $n$ is $c \cdot \log n$ for some constant $c$.

We show by induction on the input size that the contributions of the $c \log n$ terms sum up to at most $O(n)$ for the total average comparison count. Let $\mathrm{E}(A_n)$ denote the sum of the error terms in the average comparison count.

We will show that

$$\mathrm{E}(A_n) \le C \cdot n - D \ln n, \tag{14}$$

for suitable constants $C$ and $D$.

Let $D \ge c/5$. For the base case, let $n_0 \in \mathbb{N}$ and set $C$ such that $\mathrm{E}(A_n) \le C \cdot n - D \ln n$ for all $n < n_0$. As the induction hypothesis, assume that (14) holds for all $n' < n$. For the induction step, we calculate:

$$\begin{aligned}
\mathrm{E}(A_n) &= \frac{1}{\binom{n}{2}} \sum_{s+m+\ell=n} \mathrm{E}(A_n \mid s, m, \ell) \\
&\leq C \cdot n + \frac{1}{\binom{n}{2}} \sum_{s+m+\ell=n} (c \ln n - D \cdot (\ln s + \ln m + \ln \ell)) \\
&= C \cdot n + c \ln n - \frac{3}{\binom{n}{2}} \sum_{s+m+\ell=n} D \cdot \ln s \\
&= C \cdot n + c \ln n - \frac{6}{n} \sum_{1 \leq s \leq n} D \cdot \ln s
\end{aligned}$$

We use that $\sum_{i=a}^{b} f(i) \geq \int_a^b f(x)\mathrm{d}x$ for monotone functions $f$ defined on $[a, b]$ and obtain

$$\begin{aligned}
\mathrm{E}(A_n) &\leq C \cdot n + c \ln n - \frac{6D}{n} \cdot (n \ln n - n + 1) \\
&= C \cdot n + c \ln n - 6D \cdot (\ln n - 1 + 1/n).
\end{aligned}$$

An easy calculation shows that from $D \geq \frac{c}{5}$ it follows that

$$c \ln n - 6D(\ln n - 1 + 1/n) \leq -D \ln n,$$

which finishes the induction step.

Thus, the additional $O(\log n)$ terms sum up to $O(n)$ in the total average comparison count. Thus, the difference between the upper bound of $1.8n \ln n + O(n)$ derived in the proof of Theorem 3 and the exact cost is $O(n)$, and so the total average sorting cost of strategy $\mathcal{O}$ is $1.8n \ln n + O(n)$. □

## B Dual Pivot Quicksort Algorithms

### B.1 General Setup

A dual pivot quicksort method has the following general form:

**Algorithm 4 (Dual Pivot Quicksort).**

```
procedure DQS(A, left, right)
1    if $right - left \geq 1$ then
2        if $A[left] > A[right]$ then
3            swap $A[left]$ and $A[right]$
4        $p := A[left]$;
5        $q := A[right]$;
6        partition($A$, p, q, left, right, $pos_p$, $pos_q$)
7        DQS(A, left, $pos_p - 1$)
```

```
        $DQS(\text{A}, pos_p + 1, pos_q - 1)$
        $DQS(\text{A}, pos_q + 1, right)$
    end if
```

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 paper has two output variables $pos_p$ and $pos_q$ that are used to return the positions of the two pivots in the partitioned array.

As mentioned earlier, the *partition* function has two components: a *classification algorithm* and a *swap algorithm*. For each of the following algorithms, we will shortly describe these strategies before giving the actual algorithm.

## B.2 Yaroslavskiy's Partitioning Method

As mentioned in Section 4, Yaroslavskiy's algorithm makes sure that for $\ell$ large elements in the input it will compare $\ell$ elements on average 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. The swap strategy used is Meyer's swap algorithm (Algorithm 2 in Section 8).

Algorithm 5 shows the partition method of Yaroslavskiy's algorithm as studied in [15].

**Algorithm 5 (Yaroslavskiy's Partitioning Method).**

```
procedure Y-Partition($A$, $p$, $q$, $left$, $right$, $pos_p$, $pos_q$)
    $\mathtt{l} := left + 1; \mathtt{g} := right - 1; \mathtt{k} := \mathtt{l}$
    while $\mathtt{k} \leq \mathtt{g}$
        if $A[\mathtt{k}] < p$
            swap $A[\mathtt{k}]$ and $A[\mathtt{l}]$
            $\mathtt{l} := \mathtt{l} + 1$
        else
            if $A[\mathtt{k}] > q$
                while $A[\mathtt{g}] > q$ and $\mathtt{k} < \mathtt{g}$ do $\mathtt{g} := \mathtt{g} - 1$ end while
                swap $A[\mathtt{k}]$ and $A[\mathtt{g}]$
                $\mathtt{g} := \mathtt{g} - 1$
                if $A[\mathtt{k}] < p$
                    swap $A[\mathtt{k}]$ and $A[\mathtt{l}]$
                    $\mathtt{l} := \mathtt{l} + 1$
                end if
            end if
        end if
        $\mathtt{k} := \mathtt{k} + 1$
    end while
    swap $A[left]$ and $A[\mathtt{l} - 1]$
    swap $A[right]$ and $A[\mathtt{g} + 1]$
    $pos_p := \mathtt{l} - 1; pos_q := \mathtt{g} + 1;$
```

### B.3 Algorithms for the New Partitioning Method

The partitioning method 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*. We present these two classification algorithms separately. The actual partitioning method uses Algorithm 6 for the first $sz = \max(n/100, 7)$ classifications and then decides which of the two algorithms should be used for the rest of the input based on the number of small elements resp. large elements seen so far. (This is done by comparing the two variables `l` and `g` in Algorithm 6 below.)

Algorithm 6 presents a partitioning algorithm that always compares to the smaller pivot first. It uses Dijkstra's swap algorithm (Algorithm 1).

**Algorithm 6 (Simple Partitioning Method (smaller pivot first)).**

```
procedure SimplePartitionSmall($A$, $p$, $q$, $left$, $right$, $pos_p$, $pos_q$)
    l := $left + 1$; g := $right - 1$; k := l
    while k $\leq$ g
        if $A$[k] $< p$
            swap $A$[k] and $A$[l]
            l := l + 1
            k := k + 1
        else
            if $A$[k] $< q$
                k := k + 1
            else
                swap $A$[k] and $A$[g]
                g := g $-$ 1
            end if
        end if
    end while
    swap $A[left]$ and $A$[l $-$ 1]
    swap $A[right]$ and $A$[g + 1]
    $pos_p$ := l $-$ 1; $pos_q$ := g + 1
```

Algorithm 7 presents a partitioning algorithm that always compares to the larger pivot first. Note that it uses Meyer's swap algorithm and is thus not symmetrical to Algorithm 6. In our experiments, this combination of Algorithm 6 and Algorithm 7 outperformed the method that used Algorithm 6 and its symmetrical variant.

**Algorithm 7 (Simple Partitioning Method (larger pivot first)).**

```
procedure SimplePartitionLarge($A$, $p$, $q$, $left$, $right$, $pos_p$, $pos_q$)
    l := $left + 1$; g := $right - 1$; k := l
    while k $\leq$ g
        while $A$[g] $> q$
            g := g - 1
```

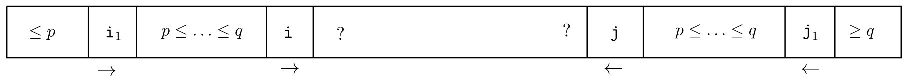

**Fig. 5.** An intermediate partitioning step in Sedgewick's algorithm.

```
        end while
        while $A[\mathtt{k}] < q$
            if $A[\mathtt{k}] < p$
                swap $A[\mathtt{k}]$ and $A[\mathtt{l}]$
                $\mathtt{l} := \mathtt{l} + 1$
            end if
            $\mathtt{k} := \mathtt{k} + 1$
        end while
        if $\mathtt{g} < \mathtt{k}$
            swap $A[\mathtt{k}]$ and $A[\mathtt{g}]$
            if $A[\mathtt{k}] < p$
                swap $A[\mathtt{l}]$ and $A[\mathtt{k}]$
                $\mathtt{l} := \mathtt{l} + 1$
            $\mathtt{g} := \mathtt{g}$ - 1
        end if
        $\mathtt{k} := \mathtt{k} + 1$
    end while
    swap $A[left]$ and $A[\mathtt{l} - 1]$
    swap $A[right]$ and $A[\mathtt{g} + 1]$
    $pos_p := \mathtt{l} - 1; pos_q := \mathtt{g} + 1$
```

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

Algorithm 8 shows Sedgewick's partitioning method as studied in [10].

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 rather has two "holes" at positions $\mathtt{i}_1$ resp. $\mathtt{j}_1$ that can be filled with small resp. 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 [10]). An intermediate step in the partitioning algorithm is depicted in Figure 5.

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 $\mathtt{i}_1$. 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 $\mathtt{j}_1$. Now it exchanges the two elements at positions `i` resp. `j` and continues until `i` and `j` have met.

**Algorithm 8 (Sedgewick's Partitioning Method).**

```
procedure S-Partition($A$, $p$, $q$, $left$, $right$, $pos_p$, $pos_q$)
    $i := i_1 := left; j := j_1 := right$;
    while true
        $i := i + 1$;
        while $A[i] \le q$
            if $i \ge j$ then break outer while end if
            if $A[i] < p$ then $A[i_1] := A[i]; i_1 := i_1 + 1; A[i] := A[i_1]$ end if
            $i := i + 1$;
        end while
        $j := j - 1$;
        while $A[j] \ge p$
            if $A[j] > q$ then $A[j_1] := A[j]; j_1 := j_1 - 1; A[j] := A[j_1]$ end if
            if $i \ge j$ then break outer while end if
            $j := j - 1$;
        end while
        $A[i_1] := A[j]; A[j_1] := A[i]$;
        $i_1 := i_1 + 1; j_1 := j_1 - 1$;
        $A[i] := A[i_1]; A[j] := A[j_1]$;
    end while
    $A[i_1] := p; A[j_1] := q$;
    $pos_p := i_1; pos_q := j_1$;
```

Algorithm 9 shows an implementation of the modified partitioning strategy from Section 4. In the same way as Algorithm 8 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 9 (Sedgewick's Partitioning Method, modified).**

```
procedure S2-Partition($A$, $p$, $q$, $left$, $right$, $pos_p$, $pos_q$)
    $i := i_1 := left; j := j_1 := right$;
    while true
        $i := i + 1$;
        while true
            if $i \ge j$ then break outer while end if
            if $A[i] < p$ then $A[i_1] := A[i]; i_1 := i_1 + 1; A[i] := A[i_1]$
            else if $A[i] > q$ then break inner while end if
            $i := i + 1$;
        end while
        $j := j - 1$;
        while true
            if $A[j] > q$ then $A[j_1] := A[j]; j_1 := j_1 - 1; A[j] := A[j_1]$
            else if $A[j] < p$ then break inner while end if
            if $\texttt{i} \geq \texttt{j}$ then break outer while end if
            $\texttt{j} := \texttt{j} - 1$;
        end while
        $A[\texttt{i}_1] := A[\texttt{j}]; A[\texttt{j}_1] := A[i]$;
        $\texttt{i}_1 := \texttt{i}_1 + 1; \texttt{j}_1 := \texttt{j}_1 - 1$;
        $A[\texttt{i}] := A[\texttt{i}_1]; A[\texttt{j}] := A[\texttt{j}_1]$;
    end while
    $A[\texttt{i}_1] := p; A[\texttt{j}_1] := q$;
    $pos_p := \texttt{i}_1; pos_q := \texttt{j}_1$;
```

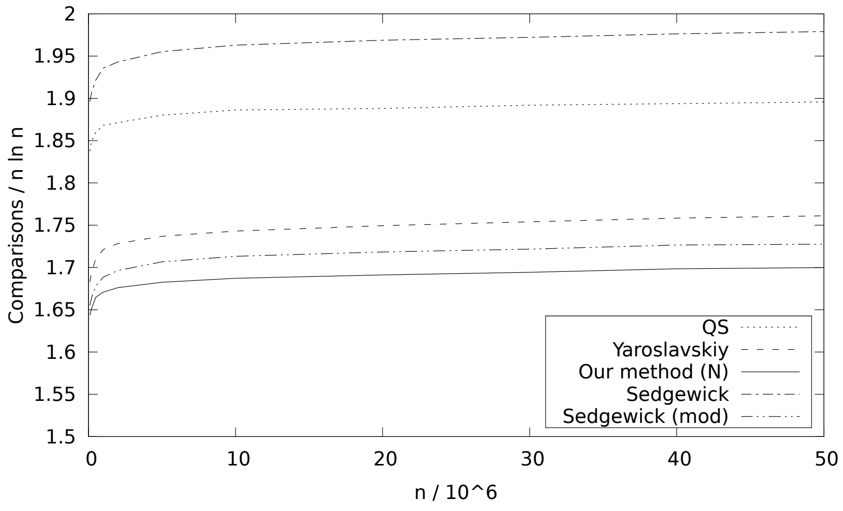

**Fig. 6.** Comparison count (scaled by $n \ln n$) needed to sort a random input of up to $n = 50\,000\,000$ integers. We compare classical quicksort, Yaroslavskiy's algorithm, the optimal algorithm from Section 5, Sedgewick's algorithm and the modified version of Sedgewick's algorithm from Section 4.

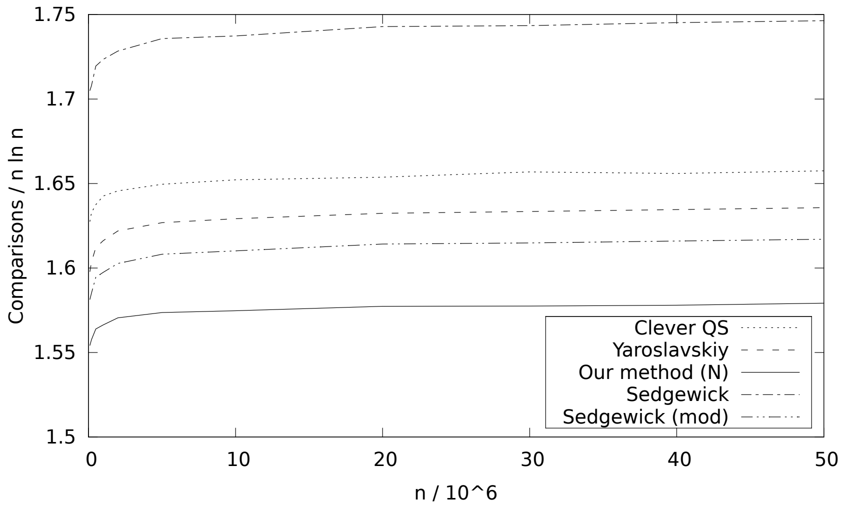

**Fig. 7.** Comparison count (scaled by $n \ln n$) needed to sort a random input of up to $n = 50\,000\,000$ integers. We compare clever quicksort—which uses the median of three elements as the pivot—, Yaroslavskiy's algorithm, the optimal algorithm from Section 5, Sedgewick's algorithm and the modified version of Sedgewick's algorithm from Section 4—all using the tertiles of a sample of five elements as the pivots.

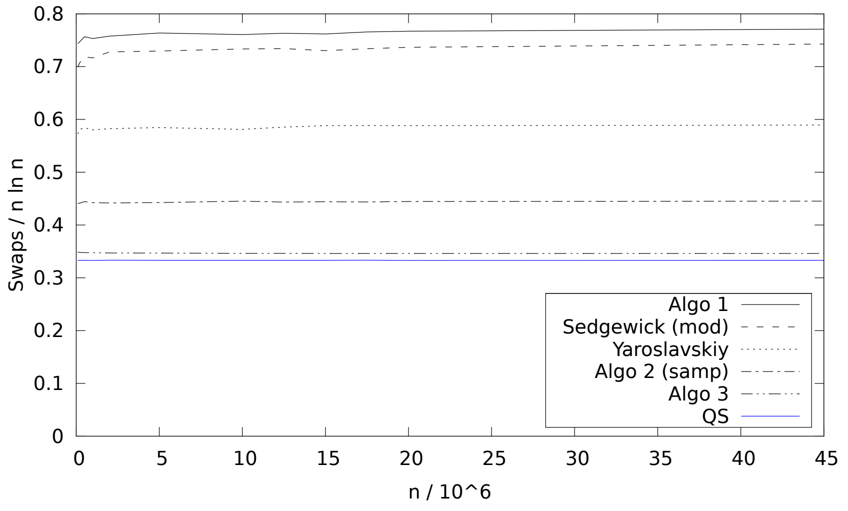

**Fig. 8.** Swap count (scaled by $n \ln n$) needed to sort a random input of up to $n = 50\,000\,000$ integers for Algorithms 1–3, Yaroslavskiy's algorithm, Sedgewick's modified algorithm (Algorithm 9) and classical quicksort. Note that for Algorithm 2 we implemented the modified version described in Section 8.2.

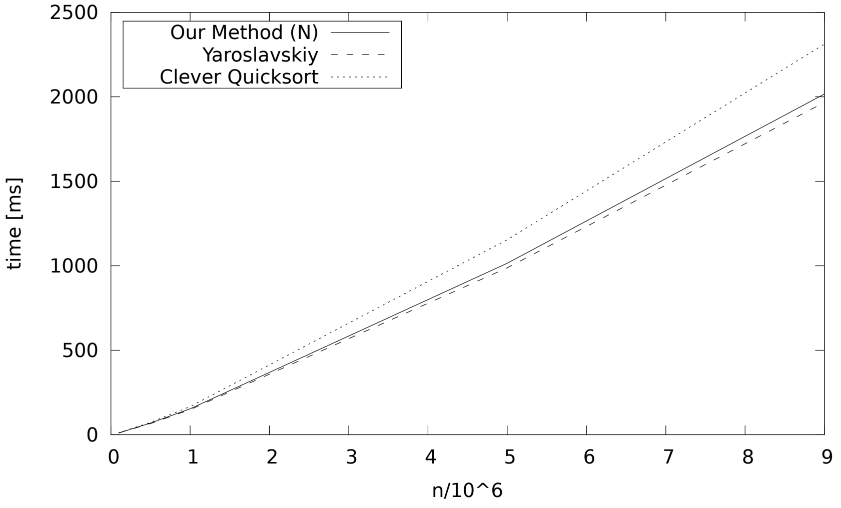

**Fig. 9.** Running time (in milliseconds) needed to sort a random permutation of $\{1, \ldots, n\}$. Clever quicksort uses the median of a sample of three elements as the median. Yaroslavskiy's algorithm and our method $\mathcal{N}$ both use the tertiles in a sample of five elements as the pivots. Running times were averaged over 1000 trials for each $n$.

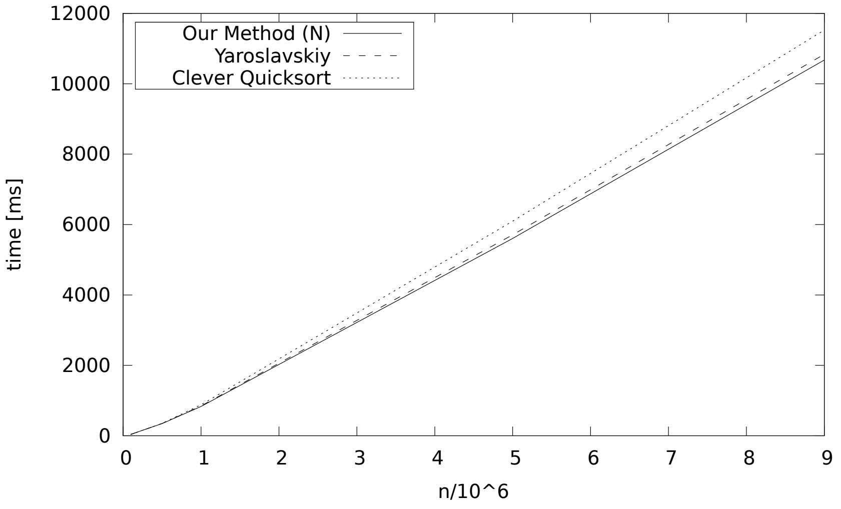

**Fig. 10.** Running time (in milliseconds) needed to sort a random set of $n$ article headers of the English Wikipedia. Running times were averaged over 1000 trials for each $n$.