How Good is Multi-Pivot Quicksort?

Martin Aumüller, Martin Dietzfelbinger, Technische Universität Ilmenau
Pascal Klaue, 3DInteractive GmbH

Multi-Pivot Quicksort refers to variants of classical quicksort where in the partitioning step k pivots are used to split the input into $k+1$ segments. For many years, multi-pivot quicksort was regarded as impractical, but in 2010 a 2-pivot approach due to Yaroslavskiy was chosen as the standard sorting algorithm in Oracle's Java. In 2014 at ALENEX, Kushagra et al. introduced an even faster algorithm that uses 3 pivots. This paper studies what possible advantages multi-pivot quicksort might offer in general. The contributions are as follows: Natural comparison-optimal algorithms for multi-pivot quicksort are devised and analyzed. The analysis shows that the benefits of using multiple pivots with respect to the average comparison count are marginal and these strategies are inferior to simpler strategies such as the well known median-of-k approach. A substantial part of the partitioning cost is caused by rearranging elements. A rigorous analysis of an algorithm for rearranging elements in the partitioning step is carried out, observing mainly how often array cells are accessed during partitioning. The algorithm behaves best if 3 or 5 pivots are used. Experiments show that this translates into good cache behavior and is closest to predicting observed running times of multi-pivot quicksort algorithms. Finally, it is studied how choosing pivots from a sample affects sorting cost.

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

General Terms: Algorithms

Additional Key Words and Phrases: Sorting, Quicksort, Multi-Pivot

ACM Reference Format:
Martin Aumüller, Martin Dietzfelbinger, and Pascal Klaue. 2015. How Good is Multi-Pivot Quicksort? *ACM Trans. Algor.* V, N, Article A (January YYYY), 49 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

sec:introduction

1. INTRODUCTION

Quicksort [Hoare 1962] is an efficient standard sorting algorithm 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, the elements in one part being smaller than or equal to the pivot, the elements in the other part being larger than or equal to the pivot, and then uses recursion to sort these parts.

In k-pivot quicksort, k elements of the input are picked and sorted to get the pivots $p_1 \leq \cdots \leq p_k$. Then the task is to partition the remaining input according to the $k+1$ *segments* or *groups* defined by the pivots. We say that an element x belongs to group $\mathrm{A}_i, 0 \leq i \leq k$, if $p_i < x < p_{i+1}$, see Fig. 1. (For ease of discussion, we set $p_0 = 0$ and

Author's addresses: M. Aumüller; M. Dietzfelbinger, Fakultät für Informatik und Automatisierung, Technische Universität Ilmenau, 98683 Ilmenau, Germany; e-mail: {martin.aumueller,martin.dietzfelbinger}@tu-ilmenau.de; P. Klaue, 3DInteractive GmbH, 98693 Ilmenau, Germany; email: pklaue@3dinteractive.de. Part of the work was done while the third author was a Master's student at Technische Universität Ilmenau.

DOI: http://dx.doi.org/10.1145/0000000.0000000

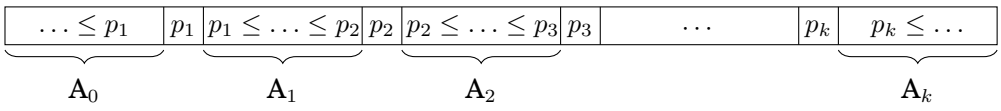

Fig. 1. Result of the partition step in k-pivot quicksort using pivots $p_1, \ldots, p_k$. fig:partition

$p_{k+1} = n + 1$.) These segments are then sorted recursively. As we will explore in this paper, using more than one pivot allows to choose from a variety of different partitioning strategies. This paper will provide the theoretical foundations to analyze these methods.

1.1. History and Related Work

Variants of classical quicksort were the topic of extensive studies, such as sampling variants [Sedgewick 1975; Martínez and Roura 2001], variants for equal keys [Sedgewick 1977], or variants for sorting strings [Bentley and Sedgewick 1997]. On the other hand, very little work has been done on quicksort variants that use more than one pivot. This is because multi-pivot quicksort was judged impractical in two independent PhD theses: In [Sedgewick 1975] Sedgewick had proposed and analyzed a dual-pivot approach that was inferior to classical quicksort in terms of the average swap count. Later, Hennequin [Hennequin 1991] studied the general approach of using $k \geq 1$ pivot elements. According to [Wild and Nebel 2012], he found only slight improvements with respect to the average comparison count that would not compensate for the more involved partitioning procedure.

Everything changed in 2009 when a 2-pivot quicksort algorithm due to Yaroslavskiy was introduced in Oracle's *Java 7*. Wild and Nebel (joined by Neininger in the full version) [2012; 2015] analyzed this algorithm and showed that it uses $1.9n \ln n + O(n)$ comparisons and $0.6n \ln n + O(n)$ swaps on average to sort a random input if two arbitrary elements are chosen as the pivots. Thus, this 2-pivot approach turned out to improve on classical quicksort—which makes $2n \ln n + O(n)$ comparisons and $0.33..n \ln n + O(n)$ swaps on average—w. r. t. the average comparison count. However, the swap count was negatively affected by using two pivots, which had also been observed for another dual-pivot quicksort algorithm in [Sedgewick 1975]. Aumüller and Dietzfelbinger [2013; 2015] showed a lower bound of $1.8n \ln n + O(n)$ comparisons on average for 2-pivot quicksort algorithms. They devised natural 2-pivot algorithms that achieved this lower bound. The key to understanding what is going on here is to note that one can improve the comparison count by deciding in a clever way with which one of the two pivots a new element should be compared first. While optimal algorithms with respect to the average comparison count are simple to implement, they must either count frequencies or need to sample a small part of the input, which renders them not competitive with Yaroslavskiy's algorithm with respect to running time when key comparisons are cheap. Moreover, Aumüller and Dietzfelbinger [2015] proposed a 2-pivot algorithm which makes $2n \ln n + O(n)$ comparisons and $0.6n \ln n + O(n)$ swaps on average—no improvement over classical quicksort in both cost measures—, but behaves very good in practice. Hence, the running time improvement of a 2-pivot quicksort approach could not be explained conclusively in these works.

Very recently, Kushagra et al. [2014] proposed a novel 3-pivot quicksort approach. Their algorithm compares a new element with the middle pivot first, and then with one of the two others. While the general idea of this algorithm had been known before (see, e. g., [Hennequin 1991; Tan 1993]), they provided a smart way of exchanging elements. Building on the work of LaMarca and Ladner [1999], they showed theoretically that their algorithm is more cache efficient than classical quicksort and Yaroslavskiy's algorithm. They reported on experiments that gave reason to believe that the improvements

of multi-pivot quicksort algorithms with respect to running times are due to their better cache behavior. They also reported from experiments with a seven-pivot algorithm, which ran more slowly than their three-pivot algorithm. We will describe how their (theoretical) arguments generalize to quicksort algorithms that use more than three pivots. In connection with the running time experiments from Section 9, this allows us to make more accurate predictions than [Kushagra et al. 2014] about the influence of cache behavior to running time. One result of this study will be that it is not surprising that their seven-pivot approach is slower, because it has worse cache behavior than three- or five-pivot quicksort algorithms using a specific partitioning strategy.

In implementations of quicksort and dual-pivot quicksort, pivots are usually taken from a small sample of elements. For example, the median in a sample of size $2k+1$ is the standard way to choose the pivot in classical quicksort. Often this sample contains only a few elements, say 3 or 5. The first theoretical analysis of this strategy is due to van Emden [1970]. Martínez and Roura [2001] settled the exact analysis of the leading term of this strategy in 2001. In practice, other pivot sampling strategies were applied successfully as well, such as the "ninther" variant from [Bentley and McIlroy 1993]. In the implementation of Yaroslavskiy's algorithm in Oracle's Java 7, the second- and fourth-largest element in a sample of size five are chosen as pivots. The exact analysis of (optimal) sampling strategies for Yaroslavskiy's algorithm is due to Nebel et al. [2015]. Interestingly, for Yaroslavskiy's algorithm it is not optimal to choose as pivots the tertiles of the sample; indeed, asymmetric choices are superior from a theoretical point of view. Moreover, it is shown there that—in contrast to classical quicksort with the median of $2k+1$ strategy—it is impossible to achieve the lower bound for comparison-based sorting algorithms using Yaroslavskiy's algorithm. Aumüller and Dietzfelbinger [2015] showed later that this is not an inherent drawback of dual-pivot quicksort. Other strategies, such as always comparing with the larger pivot first, make it again possible to achieve this lower bound. For more than two pivots, Hennequin [1991] was again the first to study how pivot sampling affects the average comparison count when a "most-balanced" comparison tree is used in each classification, see [Hennequin 1991, Tableau D.3].

1.2. Contributions

The main contributions of the present paper are as follows: (i) In the style of [Aumüller and Dietzfelbinger 2013], we study how the average comparison count of an arbitrary k-pivot quicksort algorithm can be calculated. Moreover, we show a lower bound for k-pivot quicksort and devise natural algorithms that achieve this lower bound. It will turn out that the partitioning procedures become complicated and the benefits obtained by minimizing the average comparison count are only minor. In brief, optimal k-pivot quicksort cannot improve on simple and well-studied strategies such as classical quicksort using the median-of-k strategy. Compared with the study of 2-pivot algorithms in [Aumüller and Dietzfelbinger 2013], the results generally carry over to the case of using $k \geq 3$ pivots. However, the analysis becomes more involved, and we were not able to prove tight asymptotic bounds as in the 2-pivot case. The interested reader is invited to read [Aumüller and Dietzfelbinger 2015] to get acquainted with the ideas underlying the general analysis. (ii) Leaving key comparisons aside, we study the problem of rearranging the elements to actually partition the input. We devise a natural generalization of the partitioning algorithms used in classical quicksort, Yaroslavskiy's algorithm, and the three-pivot algorithm of [Kushagra et al. 2014] to solve this problem. The basic idea is that as in classical quicksort there exist two pointers which scan the array from left to right and right to left, respectively, and the partitioning process stops when the two pointers meet. Misplaced elements are moved with the help of $k-1$ additional pointers that store starting points of special array segments. We study this

Checken: Gut so?

algorithm with regard to the average number of scanned elements (see Section 7 or [Nebel et al. 2015] for the definition),the average number of writes into array cells, and the average number of assignments necessary to rearrange the elements. Interestingly, while moving elements around becomes more complicated during partitioning, this algorithm scans fewer array cells than classical quicksort for certain (small) pivot numbers. We will see that 3- and 5-pivot quicksort algorithms visit the fewest array cells, and that this translates directly into good cache behavior and corresponds to differences in running time in practice. In brief, we provide strong evidence that the running time improvements of multi-pivot quicksort are largely due to its better cache behavior (as conjectured by Kushagra et al. [2014]), and that no benefits are to be expected from using more than 5 pivots. In the same flavor we give an analysis for two algorithms from the literature and show that they benefit from an increasing number of pivots. However, they have to store the result of a first classification step and thus require additional space. (iii) We analyze sampling strategies for multi-pivot quicksort algorithms with respect to comparisons and scanned elements. We will show that for each fixed order in which elements are compared to pivots there exist pivot choices which yield a comparison-optimal multi-pivot quicksort algorithm. When considering scanned elements there is one optimal pivot choice. Combining comparisons and scanned elements, the analysis provides a candidate for the order in which elements should be compared to pivots that has not been studied in previous attempts like [Hennequin 1991; Iliopoulos 2014]. We will now discuss the approach taken in the present paper in more detail.

1.3. Outline

In the analysis of quicksort, the analysis of one particular partitioning step with respect to a specific cost measure, e. g., the number of comparisons (or assignments, or array accesses), makes it possible to precisely analyze the cost over the whole recursion. In Hennequin's thesis [1991] the connection between partitioning cost and overall cost for quicksort variants with more than one pivot has been analyzed in detail. The result relevant for us is that if k pivots are used and the (average) partitioning cost for n elements is $a \cdot n + O(1)$, for a constant a, then the average cost for sorting n elements is

$$\frac{1}{H_{k+1} - 1} \cdot a \cdot n \ln n + O(n), \tag{1}$$

eq:1

where H_{k+1} denotes the $(k+1)$st harmonic number. In Section 2, we will use the continuous Master theorem from [Roura 2001] to prove a more general result for partitioning cost $a \cdot n + O(n^{1-\varepsilon})$. Throughout the present paper all that interests us is the constant factor with the leading term. (Of course, for real-life n the lower order term can have a big influence on the cost measure.)

For the purpose of the analysis, we will consider the input to be a random permutation of the integers $\{1, \dots, n\}$. Recall that an element x belongs to group $\mathrm{A}_i, 0 \le i \le k$, if $p_i < x < p_{i+1}$ (see Fig. 1), where we set $p_0 = 0$ and $p_{k+1} = n + 1$. When focusing on a specific cost measure we can often leave aside certain aspects of the partitioning process. For example, in the study of the average comparison count of an arbitrary k-pivot quicksort algorithm, we will only focus on *classifiying* the elements into their respective groups $\mathrm{A}_0, \dots, \mathrm{A}_k$, and omit rearranging these elements to produce the actual partition. When focusing on the average swap count (or the average number of assignments necessary to move elements around), we might just assume that the input is already classified and that the problem is just to rearrange the elements to obtain the actual partition.

In terms of classifying the elements into groups $\mathrm{A}_0, \dots, \mathrm{A}_k$, the most basic operation is the classification of a single element. This is done by comparing the element against

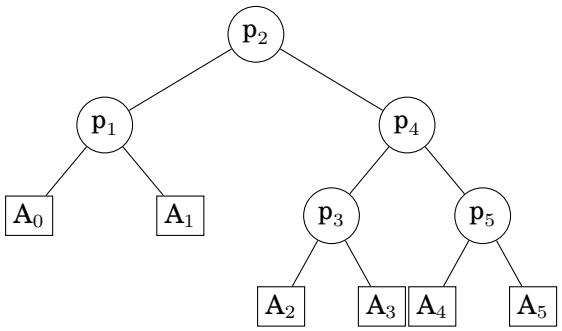

Fig. 2. A comparison tree for 5 pivots.

fig:comparison:tree

the pivots in some order. This order is best visualized using a comparison tree, which is a binary search tree with $k+1$ leaves labeled $\mathrm{A}_0, \ldots, \mathrm{A}_k$ from left to right and k inner nodes labeled $\mathrm{p}_1, \ldots, \mathrm{p}_k$ according to inorder traversal. (Such a tree for $k = 5$ is depicted in Figure 2.) Assume a comparison tree λ is given, and pivots $p_1, \ldots, p_k$ have been chosen. Then a given non-pivot element determines a search path in λ in the usual way; its classification can be read off from the leaf at the end of the path. If the input contains a_h elements of group A_h, for $0 \le h \le k$, the *cost* $\mathrm{cost}^\lambda(a_0, \ldots, a_k)$ of a comparison tree λ is the sum over all j of the depth of the leaf labeled A_j multiplied with a_j, i. e., the total number of comparisons made when classifying the whole input using λ. A classification algorithm then just defines which comparison tree is to be used for the classification of an element based on the outcome of the previous classifications. The first main result of our paper—presented in Section 3—is that in order to (approximately) determine the average comparison count for partitioning given $p_1, \ldots, p_k$ we only have to find out how many times on average each comparison tree is used by the specific algorithm. The average cost of the tree is then this average number multiplied with the cost of the tree for the pivot choice. Summing this cost over all trees gives us the average comparison count for this particular pivot choice up to lower order terms. Averaging over all pivot choices then gives the average comparison count for the classification. Section 4 applies this result by discussing different classification strategies for 3-pivot quicksort.

Besser?

In Section 5, we will show that there exist two very natural comparison-optimal strategies. The first strategy counts the number of elements $a'_0, \ldots, a'_k$ classified to groups $\mathrm{A}_0, \ldots, \mathrm{A}_k$, respectively, after the first i classifications. The comparison tree used in the $(i+1)$st classification is then just a comparison tree with minimum cost w. r. t. $(a'_0, \ldots, a'_k)$. The second strategy uses an arbitrary comparison tree for the first $n^{3/4}$ classifications, then computes a cost-minimal comparison tree for the group sizes seen in that sample, and uses this tree in each of the remaining classifications.

A full analysis of optimal versions of k-pivot quicksort does not seem possible at present for $k \ge 4$. In Section 6, we resort to estimates for the cost of partitioning based on experiments to estimate coefficients for average comparison counts for larger k. The results show that the improvements given by comparison-optimal k-pivot quicksort can be achieved in much simpler ways, e. g., by combining classical quicksort with the median-of-k pivot sampling technique. Moreover, while choosing an optimal comparison tree for fixed segment sizes is a simple application of dynamic programming, for large k the time needed for the computation renders optimal k-pivot quicksort useless with respect to running time.

Beginning with Section 7, we will follow a different approach, which we hope helps in understanding factors different from comparison counts that determine the running time of multi-pivot quicksort algorithms. We restrict ourselves to use some fixed

comparison tree for each classification, and think only about moving elements around in a swap-efficient or cache-efficient way. At the first glance, it is not clear why more pivots should help. Intuitively, the more segments we have, the more work we have to do to move elements to the segments, because we are much more restrictive on where an element should be placed. However, we save work in the recursive calls, since the denominator in (1) gets larger as we use more pivots. (Intuitively, the depth of the recursion tree decreases.) So, while the partitioning cost increases, the total sorting cost could actually decrease. We will devise an algorithm that builds upon the "crossing pointer technique" of classical quicksort. In brief, two pointers move towards each other as in classical quicksort. Misplaced elements are directly moved to some temporary array segment which holds elements of that group. This is done with the help of additional pointers. Moving misplaced elements is in general not done using swaps, but rather by moving elements in a cyclic fashion. Our cost measure for this algorithm is the number of *scanned elements*, i. e., the sum over all array cells of how many pointers accessed this array cell during partitioning and sorting. For the average number of scanned elements, it turns out that there is an interesting balance between partitioning cost and total sorting cost. In fact, in this cost measure, the average cost drastically decreases from using one pivot to using three pivots, there is almost no difference between 3- and 5-pivot quicksort, and for larger pivot numbers the average cost increases again. Interestingly, with respect to two other cost measures that look quite similar we get higher cost as the number of pivots increases. At the end of Section 7 we study two algorithms from the literature [McIlroy et al. 1993; Sanders and Winkel 2004] that can be used to rearrange elements. Both algorithms work in a two-pass fashion: in a first pass, they find out the sizes of the element groups in the input; in a second pass they use this information to make an in-place permutation of the input or allocate a new array to rearrange the input. Both of these algorithms have decreasing cost as the number of pivots increases and we discuss some natural limitations of the architecture, e. g., the size of caches, that put an upper bound on what pivot numbers still allow for efficient algorithms.

In Section 8 we turn our attention to the effect of choosing pivots from a (small) sample of elements. Building on the theoretical results regarding comparisons and scanned elements from before, it is rather easy to develop formulae to calculate the average number of comparisons and the average number of scanned elements when pivots are chosen from a small sample. Example calculations demonstrate that the cost in both measures can be decreased by choosing pivots from a small (fixed-sized) sample. Interestingly, the best pivot choices do not balance subproblem sizes but tend to make the middle groups, i. e., groups A_p with p close to $k/2$, larger. To get an idea what optimal sampling strategies should look like, we consider the setting that we can choose pivots of a given rank for free, and we are interested in the ranks that minimize the specific cost measure. Our first result in this setting shows that for every fixed comparison tree it is possible to choose the pivots in such a way that on average we need at most $1.4426..n \ln n + O(n)$ comparisons to sort the input, which is optimal. As a second result, we identify a particular pivot choice that minimizes the average number of scanned elements. In contrast to the results of the previous section we show that with these pivot choices, the average number of scanned elements decreases with a growing number of pivots.

Abschnitt 8 anpassen?

At the end of this paper, we report on experiments carried out to find if the theoretical cost measures are correlated to observed running times in practice. To this end, we implemented k-pivot quicksort variants for many different pivot numbers and compared them with respect to their running times. In brief, these experiments will confirm what has been conjectured in [Kushagra et al. 2014]: running times of quicksort algorithms are best predicted using a cost measure related to cache misses in the CPU.

sec:setup

2. SETUP AND GROUNDWORK

We assume that the input is a random permutation $(e_1, \ldots, e_n)$ of $\{1, \ldots, n\}$. If $n \leq k$, sort the input directly. For $n > k$, sort the first k elements such that $e_1 < e_2 < \ldots < e_k$ and set $p_1 = e_1, \ldots, p_k = e_k$. In the *partition step*, the remaining $n-k$ elements are split into $k+1$ *groups* $\mathrm{A}_0, \ldots, \mathrm{A}_k$, where an element x belongs to group A_h if $p_h < x < p_{h+1}$. (For the ease of discussion, we set $p_0 = 0$ and $p_{k+1} = n+1$.) The groups $\mathrm{A}_0, \ldots, \mathrm{A}_k$ are then sorted recursively. We never compare two non-pivot elements against each other. This preserves the randomness in the groups $\mathrm{A}_0, \ldots, \mathrm{A}_k$. In the remainder of this paper, we identify group sizes by $a_i := |A_i| = p_{i+1} - p_i - 1$ for $i \in \{0, \ldots, k\}$. In the first sections, we focus on analyzing the average comparison count. Let $k \geq 1$ be fixed. Let C_n denote the random variable which counts the comparisons being made when sorting an input of length n, and let P_n be the random variable which counts the comparisons made in the first partitioning step. The average comparison count of k-pivot quicksort clearly obeys the following recurrence, for $n \geq k$:

$$\mathrm{E}(C_n) = \mathrm{E}(P_n) + \frac{1}{\binom{n}{k}} \sum_{a_0 + \cdots + a_k = n-k} (\mathrm{E}(C_{a_0}) + \cdots + \mathrm{E}(C_{a_k})).$$

For $n < k$ we assume cost 0. We now collect terms with a common factor $\mathrm{E}(C_\ell)$, for $0 \leq \ell \leq n-k$. To this end, fix $j \in \{0, \ldots, k\}$ and $\ell \in \{0, \ldots, n-k\}$ and assume that $a_j = \ell$. By a standard argument, there are exactly $\binom{n-\ell-1}{k-1}$ ways to choose the other segment sizes $a_i, i \neq j$, such that $a_0 + \cdots + a_k = n-k$. (Note the equivalence between segment sizes and binary strings of length $n-\ell-1$ with exactly $k-1$ ones.) Thus, we conclude that

$$\mathrm{E}(C_n) = \mathrm{E}(P_n) + \frac{k+1}{\binom{n}{k}} \sum_{\ell=0}^{n-k} \binom{n-\ell-1}{k-1} \mathrm{E}(C_\ell), \tag{2}$$

eq:k:pivot:recurrence

which was also observed in [Iliopoulos 2014]. (This generalizes the well known formula $\mathrm{E}(C_n) = n - 1 + 2/n \cdot \sum_{0 \leq \ell \leq n-1} \mathrm{E}(C_\ell)$ for classical quicksort and the formulas for $k = 2$ from, e. g., [Aumüller and Dietzfelbinger 2013; Wild and Nebel 2012] and $k = 3$ from [Kushagra et al. 2014].) For partitioning cost of $a \cdot n + O(n^{1-\varepsilon})$, for constants a and $\varepsilon > 0$, this recurrence has the following solution.

THEOREM 2.1. *Let $\mathcal{A}$ be a k-pivot quicksort algorithm that for each subarray of length n has partitioning cost $\mathrm{E}(P_n) = a \cdot n + O(n^{1-\varepsilon})$ for a constant $\varepsilon > 0$. Then*

$$\mathrm{E}(C_n) = \frac{1}{H_{k+1} - 1} \cdot an \ln n + O(n), \tag{3}$$

eq:k:pivot:recurrence

t:recurrence:solution

where $H_{k+1} = \sum_{i=1}^{k+1} (1/i)$ is the $(k+1)$st harmonic number.

PROOF. By linearity of expectation we may solve the recurrence for partitioning cost $\mathrm{E}(P_{1,n}) = a \cdot n + O(1)$ and $\mathrm{E}(P_{2,n}) = O(n^{1-\varepsilon})$ separately. To solve for cost $\mathrm{E}(P_{1,n})$ we may apply (1). The influence of partitioning cost $\mathrm{E}(P_{2,n})$ over the whole recursion can be analyzed using the continuous Master theorem from [Roura 2001]. These calculations can be found in Appendix A. □

When focusing only on the average comparison count, it suffices to study the *classification problem*: Given a random permutation $(e_1, \ldots, e_n)$ of $\{1, \ldots, n\}$, choose the pivots $p_1, \ldots, p_k$ and classify each of the remaining $n-k$ elements as belonging to one of the groups $\mathrm{A}_0, \ldots, \mathrm{A}_k$.

Algorithmically, the classification of a single element x with respect to the pivots $p_1, \ldots, p_k$ is done by using a *comparison tree* λ. A comparison tree is a binary search tree, where the leaf nodes are labeled $\mathrm{A}_0, \ldots, \mathrm{A}_k$ from left to right and the inner nodes are labeled $\mathrm{p}_1, \ldots, \mathrm{p}_k$ in inorder. Figure 2 depicts a comparison tree for 5 pivots. Classifying an element then means searching for this element in the search tree. The classification of the element is the label of the leaf reached in that way. If x belongs to group A_h, the depth $\mathrm{depth}_\lambda(\mathrm{A}_h)$ of the label A_h in λ is the number of comparisons made.

Satz checken.

A *classification strategy* is formally described as a *classification tree* as follows. A classification tree is a $(k+1)$-way tree with a root and $n-k$ levels of inner nodes as well as one leaf level. Each inner node v has two labels: an index $i(v) \in \{k+1, \ldots, n\}$, and a comparison tree $\lambda(v)$. The element $e_{i(v)}$ is classified using the comparison tree $\lambda(v)$. The $k+1$ edges out of a node are labeled $0, \ldots, k$, resp., representing the outcome of the classification as belonging to group $\mathrm{A}_0, \ldots, \mathrm{A}_k$, respectively. On each of the $(k+1)^{n-k}$ paths each index from $\{k+1, \ldots, n\}$ occurs exactly once. An input $(e_1, \ldots, e_n)$ determines a path in the classification tree in the obvious way: sort the pivots, then use the classification tree to classify $e_{k+1}, \ldots, e_n$. The classification of the input can then be read off from the nodes and edges along the path from the root to a leaf in the classification tree.

To fix some more notation, for each node v, and for $h \in \{0, \ldots, k\}$, we let a_h^v be the number of edges labeled "h" on the path from the root to v. Furthermore, let $C_{h,i}$ denote the random variable which counts the number of elements classified as belonging to group A_h, for $h \in \{0, \ldots k\}$, in the first i levels, for $i \in \{0, \ldots, n-k\}$, i. e., $C_{h,i} = a_h^v$ when v is the node on level i of the classification tree reached for an input. In many proofs, we will need that $C_{h,i}$ is not far away from its expectation $a_h/(n-k-i)$. This would be a trivial consequence of the Chernoff bound if the classification of elements were independent. However, the probabilities of classifying elements to a specific group change according to classifications made before. We will use the *method of averaged bounded differences* to show concentration despite dependencies between tests.

LEMMA 2.2. *Let the pivots $p_1, \ldots, p_k$ be fixed. Let $C_{h,i}$ be defined as above. Then for each h with $h \in \{0, \ldots, k\}$ and for each i with $1 \le i \le n-k$ we have that*

$$\Pr\Big(|C_{h,i} - \mathbf{E}(C_{h,i})| > n^{2/3}\Big) \le 2\exp\Big(-n^{1/3}/2\Big).$$

oncentration:k:pivots

PROOF. Fix an arbitrary $h \in \{0, \ldots, k\}$. Define the indicator random variable

$$X_j = [\text{the element classified in level } j \text{ belongs to group } \mathrm{A}_h].$$

Of course, $C_{h,i} = \sum_{1 \le j \le i} X_j$. We let

$$c_j := |\mathbf{E}(C_{h,i} \mid X_1, \ldots, X_j) - \mathbf{E}(C_{h,i} \mid X_1, \ldots, X_{j-1})|.$$

Rechnung anpassen.

Using linearity of expectation we may calculate

$$
\begin{aligned}
c_j &= \bigl|\mathbf{E}(C_{h,i} \mid X_1,\dots,X_j) - \mathbf{E}(C_{h,i} \mid X_1,\dots,X_{j-1})\bigr| \\
&= \left| X_j - \mathbf{E}(X_j \mid X_1,\dots,X_{j-1}) + \sum_{\ell=j+1}^{i} \bigl(\mathbf{E}(X_\ell \mid X_1,\dots,X_j) - \mathbf{E}(X_\ell \mid X_1,\dots,X_{j-1})\bigr)\right| \\
&= \left| X_j - \frac{a_h - C_{h,j-1}}{n-j-1} + (i-j)\left(\frac{a_h - C_{h,j}}{n-j-2} - \frac{a_h - C_{h,j} + X_j}{n-j-1}\right)\right| \\
&= \left| X_j\left(1 - \frac{i-j}{n-j-1}\right) - \frac{a_h - C_{h,j-1}}{n-j-1} + \frac{(i-j)(a_h - C_{h,j})}{(n-j-2)(n-j-1)}\right| \\
&\le \left| X_j\left(1 - \frac{i-j}{n-j-1}\right) - \frac{a_h - C_{h,j} + X_j}{n-j-1} + \frac{a_h - C_{h,j}}{n-j-1}\right| \\
&= \left| X_j\left(1 - \frac{i-j-1}{n-j-1}\right)\right| \le 1.
\end{aligned}
$$

We use the following bound known as the method of averaged bounded differences (see [Dubhashi and Panconesi 2009, Theorem 5.3]):

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

This yields

$$
\Pr\Bigl(|C_{h,i} - \mathbf{E}(C_{h,i})| > n^{2/3}\Bigr) \le 2\exp\left(\frac{-n^{4/3}}{2i}\right),
$$

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

sec:act

3. THE AVERAGE COMPARISON COUNT FOR PARTITIONING

In this section, we will obtain a formula for the average comparison count of an arbitrary classification strategy. We make the following observations for all classification strategies: We need $k \log k = O(1)$ comparisons to sort $e_1,\dots,e_k$, i. e., to determine the k pivots $p_1,\dots,p_k$ in order. If an element x belongs to group A_i, it must be compared to p_i and p_{i+1}. (Of course, no real comparison takes place against p_0 and p_{k+1}.) On average, this leads to $2(1-1/(k+1))(n-k) + O(1)$ comparisons—regardless of the actual classification strategy.

For the following paragraphs, we fix a classification strategy, i. e., a classification tree T. Furthermore, we let v be an arbitrary inner node of T.

If $e_{i(v)}$ belongs to group A_h then exactly $\mathrm{depth}_{\lambda(v)}(\mathrm{A}_h)$ comparisons are made to classify this element. We let C_v^T denote the number of comparisons that take place in node v during classification. Let P_n^T be the random variable that counts the number of comparisons being made when classifying an input sequence $(e_1,\dots,e_n)$ using T, i. e., $P_n^T = \sum_{v\in T} C_v^T$. For the average classification cost $\mathbf{E}(P_n^T)$ we get:

Use $\mathbf{p}$ and $\mathbf{a}$ instead of $a_0,\dots,a_k$?

$$
\mathbf{E}(P_n^T) = \frac{1}{\binom{n}{k}} \sum_{1\le p_1<p_2<\cdots<p_k\le n} \mathbf{E}(P_n^T \mid p_1,\dots,p_k).
$$

We define $p^v_{p_1,\ldots,p_k}$ as the probability that node v is reached if the pivots are $p_1,\ldots,p_k$. We may write:

$$\begin{aligned}\mathbf{E}(P_n^T \mid p_1,\ldots,p_k) &= \sum_{v\in T} \mathbf{E}(C_v^T \mid p_1,\ldots,p_k)\\ &= \sum_{v\in T} p^v_{p_1,\ldots,p_k}\cdot \mathbf{E}(C_v^T \mid p_1,\ldots,p_k,\ v \text{ reached}). \qquad (4)\end{aligned}$$

eq:51000

For a comparison tree λ and group sizes $a'_0,\ldots,a'_k$, we define the *cost* of λ on these group sizes as the number of comparisons it makes for classifying an input with these group sizes, i. e.,

$$\text{cost}^\lambda(a'_0,\ldots,a'_k) = \sum_{0\le i\le k} \text{depth}_\lambda(\mathrm{A}_i)\cdot a'_i.$$

Furthermore, we define its average cost $c^\lambda_{\text{avg}}(a'_0,\ldots,a'_k)$ as follows:

$$c^\lambda_{\text{avg}}(a'_0,\ldots,a'_k) := \frac{\text{cost}^\lambda(a'_0,\ldots,a'_k)}{\sum_{0\le i\le k} a'_i}. \qquad (5)$$

eq:30014

Under the assumption that node v is reached and that the pivots are $p_1,\ldots,p_k$, the probability that the element $e_{i(v)}$ belongs to group A_h is exactly $(a_h - a^v_h)/(n-k-\text{level}(v))$, for each $h\in\{0,\ldots,k\}$. (Note that this means that the order in which elements are classified is arbitrary, so that we could actually use some fixed ordering.) Summing over all groups, we get

$$\mathbf{E}(C_v^T \mid p_1,\ldots,p_k,\ v \text{ reached}) = c^{\lambda(v)}_{\text{avg}}(a_0-a^v_0,\ldots,a_k-a^v_k).$$

Plugging this into (4) gives

$$\mathbf{E}(P_n^T \mid p_1,\ldots,p_k) = \sum_{v\in T} p^v_{p_1,\ldots,p_k}\cdot c^{\lambda(v)}_{\text{avg}}(a_0-a^v_0,\ldots,a_k-a^v_k). \qquad (6)$$

eq:50000

Let Λ_k be the set of all possible comparison trees. For each $\lambda\in\Lambda_k$, we define the random variable F^λ that counts the number of times λ is used during classification. For given $p_1,\ldots,p_k$, and for each $\lambda\in\Lambda_k$, we let

$$f^\lambda_{p_1,\ldots,p_k} := \mathbf{E}(F^\lambda \mid p_1,\ldots,p_k) = \sum_{\substack{v\in T\\ \lambda(v)=\lambda}} p^v_{p_1,\ldots,p_k}$$

denote the average number of times comparison tree λ is used in T under the condition that the pivots are $p_1,\ldots,p_k$.

Now, if it was decided in each step by independent random experiments with the correct expectation $a_h/(n-k)$, for $0\le h\le k$, whether an element belongs to group A_h, it would be clear that for each $\lambda\in\Lambda_k$ the contribution of λ to the average classification cost is $f^\lambda_{p_1,\ldots,p_k}\cdot c^\lambda_{\text{avg}}(a_0,\ldots,a_k)$. This intuition can be proven to hold for all classification trees, except that one gets an additional $O(n^{1-\varepsilon})$ term due to dependencies between classifications.

LEMMA 3.1. *Let the pivots $p_1,\ldots,p_k$ be fixed. Let T be a classification tree. Then there exists a constant $\varepsilon>0$ such that*

$$\mathbf{E}(P_n^T) = \frac{1}{\binom{n}{k}} \sum_{1\le p_1<p_2<\cdots<p_k\le n} \ \sum_{\lambda\in\Lambda_k} f^\lambda_{p_1,\ldots,p_k}\cdot c^\lambda_{\text{avg}}(a_0,\ldots,a_k) + O(n^{1-\varepsilon}).$$

verage:partition:cost

PROOF. Fix the set of pivots $p_1, \ldots, p_k$. The calculations start from re-writing (6) in the following form:

$$\begin{aligned}
\mathbf{E}(P_n^T \mid p_1, \ldots, p_k) &= \sum_{v \in T} p^v_{p_1,\ldots,p_k} \cdot c^{\lambda(v)}_{\text{avg}}(a_0 - a_0^v, \ldots, a_k - a_k^v) \\
&= \sum_{v \in T} p^v_{p_1,\ldots,p_k} \cdot c^{\lambda(v)}_{\text{avg}}(a_0, \ldots, a_k) - \\
&\qquad \sum_{v \in T} p^v_{p_1,\ldots,p_k} \Big(c^{\lambda(v)}_{\text{avg}}(a_0, \ldots, a_k) - c^{\lambda(v)}_{\text{avg}}(a_0 - a_0^v, \ldots, a_k - a_k^v) \Big) \\
&= \sum_{\lambda \in \Lambda_k} f^{\lambda}_{p_1,\ldots,p_k} \cdot c^{\lambda}_{\text{avg}}(a_0, \ldots, a_k) - \\
&\qquad \sum_{v \in T} p^v_{p_1,\ldots,p_k} \Big(c^{\lambda(v)}_{\text{avg}}(a_0, \ldots, a_k) - c^{\lambda(v)}_{\text{avg}}(a_0 - a_0^v, \ldots, a_k - a_k^v) \Big).
\end{aligned} \tag{7}$$

eq:50002

For each node v in the classification tree, we say that v is *on track* (to the expected values) if

$$\left| c^{\lambda(v)}_{\text{avg}}(a_0, \ldots, a_k) - c^{\lambda(v)}_{\text{avg}}(a_0 - a_0^v, \ldots, a_k - a_k^v) \right| \le \frac{k^2}{n^{1/12}}.$$

Otherwise, v is called *off track*.

By considering on-track and off-track nodes in (7) separately, we may calculate

$$\begin{aligned}
\mathbf{E}(P_n^T \mid p_1, \ldots, p_k) &\le \sum_{\lambda \in \Lambda_k} f^{\lambda}_{p_1,\ldots,p_k} \cdot c^{\lambda}_{\text{avg}}(a_0, \ldots, a_k) + \sum_{\substack{v \in T \\ v \text{ is on track}}} p^v_{p_1,\ldots,p_k} \frac{k^2}{n^{1/12}} + \\
&\qquad \sum_{\substack{v \in T \\ v \text{ is off track}}} p^v_{p_1,\ldots,p_k} \Big(c^{\lambda(v)}_{\text{avg}}(a_0, \ldots, a_k) - c^{\lambda(v)}_{\text{avg}}(a_0 - a_0^v, \ldots, a_k - a_k^v) \Big) \\
&\le \sum_{\lambda \in \Lambda_k} f^{\lambda}_{p_1,\ldots,p_k} \cdot c^{\lambda}_{\text{avg}}(a_0, \ldots, a_k) + k \cdot \sum_{\substack{v \in T \\ v \text{ is off track}}} p^v_{p_1,\ldots,p_k} + O(n^{11/12}) \\
&= \sum_{\lambda \in \Lambda_k} f^{\lambda}_{p_1,\ldots,p_k} \cdot c^{\lambda}_{\text{avg}}(a_0, \ldots, a_k) + \\
&\qquad k \cdot \sum_{i=1}^{n-k} \Pr(\text{an off track node is reached on level } i) + O(n^{11/12}).
\end{aligned} \tag{8}$$

eq:50001

It remains to bound the second summand of (8). First, we obtain the general bound:

$$\begin{aligned}
&\left| c^{\lambda(v)}_{\text{avg}}(a_0, \ldots, a_k) - c^{\lambda(v)}_{\text{avg}}(a_0 - a_0^v, \ldots, a_k - a_k^v) \right| \\
&\qquad \le (k-1) \cdot \sum_{j=0}^{k} \left| \frac{a_j}{n-k} - \frac{a_j - a_j^v}{n - k - \text{level}(v)} \right| \\
&\qquad \le (k-1) \cdot (k+1) \cdot \max_{0 \le j \le k} \left\{ \left| \frac{a_j}{n-k} - \frac{a_j - a_j^v}{n - k - \text{level}(v)} \right| \right\}.
\end{aligned}$$

Thus, by definition, whenever v is an off track node, there exists $j \in \{0, \dots, k\}$ such that

$$\left| \frac{a_j}{n-k} - \frac{a_j - a_j^v}{n-k-\text{level}(v)} \right| > \frac{1}{n^{1/12}}.$$

Now consider the case that the random variables $C_{h,i}$ that counts the number of A_h-elements in the first i classifications are concentrated around their expectation, as in the statement of Lemma 2.2. This happens with very high probability, so the contributions of the other case to the average comparison count can be neglected. For each $h \in \{0, \dots, k\}$, and each level $i \in \{1, \dots, n-k\}$ we calculate

$$\left| \frac{a_h}{n-k} - \frac{a_h - C_{h,i}}{n-k-i} \right| \le \left| \frac{a_h}{n-k} - \frac{a_h(1-i/(n-k))}{n-k-i} \right| + \left| \frac{n^{2/3}}{n-k-i} \right| = \frac{n^{2/3}}{n-k-i}.$$

So, for the first $i \le n - n^{3/4}$ levels, we are with very high probability in an *on track node* on level i, because the deviation of the ideal probability $a_h/(n-k)$ of seeing an element which belongs to group A_h and the actual probability in the node reached on level i of seeing such an element 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 off track nodes is not more than $O(n^{11/12})$ to the first summand in (8). For the last $n^{3/4}$ levels of the tree, we use that the contribution of the probabilities that we reach an off track node on level i is at most 1 for a fixed level.

This shows that the second summand in (8) is $O(n^{11/12})$. The lemma now follows from averaging over all possible pivot choices. □

sec:3:pivot:quicksort

4. EXAMPLE: 3-PIVOT QUICKSORT

Here we study variants of 3-pivot quicksort algorithms in the light of Lemma 3.1. This paradigm got recent attention by the work of Kushagra et al. [2014], who provided evidence that—in practice—a 3-pivot quicksort algorithm might be faster than Yaroslavskiy's dual-pivot quicksort.

In 3-pivot quicksort, we might choose from five different comparison trees. These trees, together with their comparison cost, are depicted in Figure 3. We will study the average comparison count of three different strategies in an artificial setting: We assume, as in the analysis, that our input is a permutation of $\{1, \dots, n\}$. So, after choosing the pivots the algorithm knows the exact group sizes in advance. Transforming this strategy into a realistic one is a topic of the next section.

All considered strategies will follow the same idea: After choosing the pivots, it is checked which comparison tree has the smallest average cost for the group sizes found in the input. Then this tree is used for all classifications. Our strategies differ in respect to the set of comparison trees they can use. In the next section we will explain why deviating from such a strategy, i. e., using different trees during the classification for fixed group sizes, does not help for minimizing the average comparison count.

The symmetric strategy. In the algorithm of [Kushagra et al. 2014], the balanced comparison tree λ_2 is used for each classification. Using Lemma 3.1, we get[1]

$$\begin{aligned} \mathrm{E}(P_n) &= \frac{1}{\binom{n}{3}} \sum_{a_0+a_1+a_2+a_3=n-3} (2a_0 + 2a_1 + 2a_2 + 2a_3) + O(n^{1-\varepsilon}) \\ &= 2n + O(n^{1-\varepsilon}). \end{aligned}$$

Using Theorem 2.1, we conclude that

[1] Of course, $\mathrm{E}(P_n) = 2(n-3)$, since each classification makes exactly two comparisons.

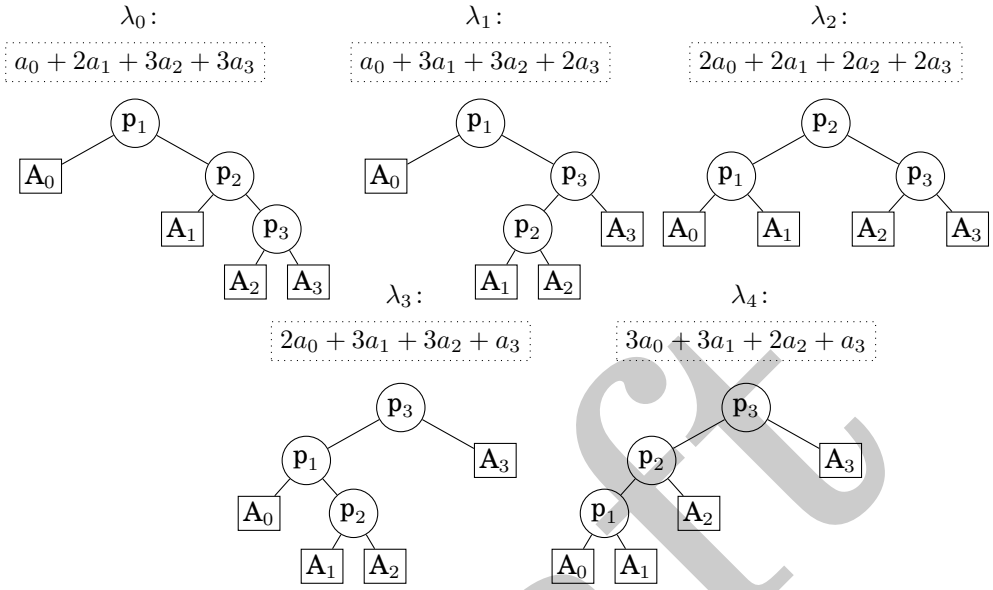

Fig. 3. The different comparison trees for 3-pivot quicksort with their comparison cost (dotted boxes, only displaying the numerator).

fig:3:pivot:comparisc

$$\mathbf{E}(C_n) = 24/13 n \ln n + O(n) \approx 1.846 n \ln n + O(n),$$

as known from [Kushagra et al. 2014]. This improves on classical quicksort ($2n \ln n + O(n)$ comparisons on average), but is worse than optimal dual-pivot quicksort ($1.8n \ln n + O(n)$ comparisons on average [Aumüller and Dietzfelbinger 2013]) or median-of-3 quicksort ($1.714n \ln n + O(n)$ comparisons on average [van Emden 1970]).

Using three trees. Here we restrict our algorithm to choose only among the comparison trees $\{\lambda_1, \lambda_2, \lambda_3\}$. The computation of a cost-minimal comparison tree is then simple: Suppose that the segment sizes are $a_0, \dots, a_3$. If $a_0 > a_3$ and $a_0 > a_1 + a_2$ then comparison tree λ_1 has minimum cost. If $a_3 \geq a_0$ and $a_3 > a_1 + a_2$ then comparison tree λ_3 has minimum cost. Otherwise λ_2 has minimum cost.

Using Lemma 3.1, the average partition cost with respect to this set of comparison trees can be calculated (using Maple®) as follows:

$$\begin{aligned}\mathbf{E}(P_n) &= \frac{1}{\binom{n}{3}} \sum_{a_0+a_1+a_2+a_3=n-3} \min\left\{{a_0+3a_1+3a_2+2a_3, 2a_0+2a_1+2a_2+2a_3, \atop 2a_0+3a_1+3a_2+1a_3}\right\} + O(n^{1-\varepsilon}) \\ &= \frac{17}{9} n + O(n^{1-\varepsilon}).\end{aligned}$$

This yields the following average comparison cost:

$$\mathbf{E}(C_n) = \frac{68}{39} n \ln n + O(n) \approx 1.744 n \ln n + O(n).$$

Using all trees. Now we let our strategies choose among all five trees. Using Lemma 3.1 and the average cost for all trees from Figure 3, we calculate (using Maple®)

$$\mathbf{E}(P_n) = \frac{1}{\binom{n}{3}} \sum_{a_0+a_1+a_2+a_3=n-3} \min\left\{\begin{smallmatrix} a_0+2a_1+3a_2+3a_3, a_0+3a_1+3a_2+2a_3, \\ 2a_0+2a_1+2a_2+2a_3, 2a_0+3a_1+3a_2+a_3 \\ 3a_0+3a_1+2a_2+a_3 \end{smallmatrix}\right\} + O(n^{1-\varepsilon})$$
$$= \frac{133}{72} n + O(n^{1-\varepsilon}).$$

This yields the following average comparison cost:

$$\mathbf{E}(C_n) = \frac{133}{78} n \ln n + O(n) \approx 1.705 n \ln n + O(n),$$

which is—as will be explained in the next section—the lowest possible average comparison count one can achieve by picking three pivots directly from the input. So, using three pivots gives a slightly lower average comparison count than quicksort using the median of three elements as the pivot.

ec:optimal:strategies

5. (ASYMPTOTICALLY) OPTIMAL CLASSIFICATION STRATEGIES

In this section we will discuss four different strategies, which will all achieve the minimal average comparison count (up to lower order terms). Two of these four strategies will be optimal but unrealistic, since they assume that after the pivots are fixed the algorithm knows the sizes of the $k+1$ different groups. The strategies work as follows: One strategy maintains the group sizes of the unclassified part of the input and chooses the comparison tree with minimum cost with respect to these group sizes. This will turn out to be the optimal classification strategy for k-pivot quicksort. To turn this strategy into an actual algorithm, we will use the group sizes of the already classified part of the input as a basis for choosing the comparison tree for the next classification. The second unrealistic strategy works like the algorithms for 3-pivot quicksort. It will use the comparison tree with minimum cost with respect to the group sizes of the input in each classification. To get an actual algorithm, we estimate these group sizes in a small sampling step. Note that these strategies are the obvious generalization of the optimal strategies for dual-pivot quicksort from [Aumüller and Dietzfelbinger 2013].

Since all these strategies need to compute cost-minimal comparison trees, this section starts with a short discussion of algorithms for this problem. Then we discuss the four different strategies.

timal:comparison:tree

5.1. Choosing an Optimal Comparison Tree

For optimal k-pivot quicksort algorithms it is of course necessary to devise an algorithm that can compute an optimal comparison tree for partition sizes $a_0, \dots, a_k$, i. e., a comparison tree that minimizes (5). It is well known that the number of binary search trees with k inner nodes equals the k-th Catalan number, which is approximately $4^k/((k+1)\sqrt{\pi k})$. Choosing an optimal comparison tree is a standard application of dynamic programming, and is known from textbooks as "choosing an optimum binary search tree", see, e. g., [Knuth 1973]. The algorithm runs in time and space $O(k^2)$.

Das kann man wohl in $O(k \log k)$ lösen. Check Link in Texfile.

5.2. The Optimal Classification Strategy and its Algorithmic Variant

Here, we consider the following strategy[2] $\mathcal{O}_k$: *Given $a_0, \dots, a_k$, the comparison tree $\lambda(v)$ is one that minimizes cost$^\lambda(a_0 - a_0^v, \dots, a_k - a_k^v)$ over all comparison trees λ.*

[2]For all strategies we just say which comparison tree is used in a given node of the classification tree. Recall that the classification order is arbitrary.

Although being unrealistic, since the exact partition sizes $a_0, \ldots, a_k$ are in general unknown to the algorithm, strategy $\mathcal{O}_k$ is the optimal classification strategy, i. e., it minimizes the average comparison count.

thm:o:k:optimal

THEOREM 5.1. *Strategy $\mathcal{O}_k$ is optimal for each k.*

PROOF. Strategy $\mathcal{O}_k$ chooses for each node v in the classification tree the comparison tree that minimizes the average cost in (6). So, it minimizes each term of the sum and thus minimizes the whole sum in (6). □

There exist other strategies whose average comparison count differs by at most $O(n^{1-\varepsilon})$ from the average comparison count of $\mathcal{O}_k$. We call such strategies *asymptotically optimal*.

Strategy $\mathcal{C}_k$ is an algorithmic variant of $\mathcal{O}_k$. It works as follows: *The comparison tree $\lambda(v)$ is one that minimizes cost$^\lambda(a_0^v, \ldots, a_k^v)$ over all comparison trees λ.*

thm:l:k:optimal

THEOREM 5.2. *Strategy $\mathcal{C}_k$ is asymptotically optimal for each k.*

PROOF. Since the average comparison count is independent of the actual order in which elements are classified, assume that strategy $\mathcal{O}_k$ classifies elements in the order $e_{k+1}, \ldots, e_n$, while strategy $\mathcal{C}_k$ classifies them in reversed order, i. e., $e_n, \ldots, e_{k+1}$. Then the comparison tree that is used by $\mathcal{C}_k$ for element e_i is the one that $\mathcal{O}_k$ is using for element e_{i+1} because both strategies use the group sizes in $(e_{i+1}, \ldots, e_n)$. Let P_i and P_i' denote the number of comparisons for the classification of the element e_{k+i} using strategy $\mathcal{O}_k$ and $\mathcal{C}_k$, respectively.

Fix some integer $i \in \{1, \ldots, n-k\}$. Suppose that the input has group sizes $a_0, \ldots, a_k$. Assume that the sequence $(e_{k+1}, \ldots, e_i)$ contains a_h' elements of group A_h for $h \in \{0, \ldots, k\}$, where $|a_h' - i \cdot a_h/(n-k)| \le n^{2/3}$ for each $h \in \{0, \ldots, k\}$. Let λ be a comparison tree with minimal cost w. r. t. $(a_0 - a_0', \ldots, a_k - a_k')$. For a random input having group sizes from above we calculate:

Rechnung anpassen.

$$
\begin{aligned}
&\left|\mathbf{E}(P_{i+1}) - \mathbf{E}(P_i')\right| \\
&= \left|c^\lambda_{\mathbf{avg}}(a_0 - a_0', \ldots, a_k - a_k') - c^\lambda_{\mathbf{avg}}(a_0', \ldots, a_k')\right| \\
&= \left|\frac{\sum_{h=0}^k \mathbf{depth}_\lambda(\mathrm{A}_h) \cdot (a_h - a_h')}{n-k-i} - \frac{\sum_{h=0}^k \mathbf{depth}_\lambda(\mathrm{A}_h) \cdot a_h'}{i}\right| \\
&\le \left|\frac{\sum_{h=0}^k \mathbf{depth}_\lambda(\mathrm{A}_h) \cdot (a_h - (i \cdot \frac{a_h}{n-k} - n^{2/3}))}{n-k-i} - \frac{\sum_{h=0}^k \mathbf{depth}_\lambda(\mathrm{A}_h) \cdot \left(i \cdot \frac{a_h}{n-k} - n^{2/3}\right)}{i}\right| \\
&\le \left|\frac{k^2 \cdot n^{2/3} + \sum_{h=0}^k \mathbf{depth}_\lambda(\mathrm{A}_h)(a_h - \frac{a_h \cdot i}{n-k}))}{n-k-i} - \frac{\sum_{h=0}^k \mathbf{depth}_\lambda(\mathrm{A}_h)\left(i \cdot \frac{a_h}{n-k}\right) - k^2 \cdot n^{2/3}}{i}\right| \\
&= \left|\frac{k^2 \cdot n^{2/3}}{n-k-i} + \frac{k^2 \cdot n^{2/3}}{i}\right|.
\end{aligned}
$$

Assume that the concentration argument of Lemma 2.2 holds. Then the difference between the average comparison count for element e_{i+1} (for $\mathcal{O}_k$) and e_i (for $\mathcal{C}_k$) is at most

$$\left|\frac{k^2 \cdot n^{2/3}}{n-k-i} + \frac{k^2 \cdot n^{2/3}}{i}\right|.$$

The difference of the average comparison count over all elements $e_i, \dots, e_j$, $i \geq n^{3/4}, j \leq n - n^{3/4}$, is then at most $O(n^{11/12})$. For elements that reside outside of this range, the difference in the average comparison count is at most $2n^{3/4} \cdot k$. Furthermore, error terms for cases where the concentration argument does not hold can be neglected because they occur with exponentially low probability. So, the total difference of the average comparison count between strategy $\mathcal{O}_k$ and strategy $\mathcal{C}_k$ is at most $O(n^{11/12})$. □

This shows that the optimal strategy $\mathcal{O}_k$ can be approximated by an actual algorithm that makes an error of up to $O(n^{11/12})$, which sums up to an error term of $O(n)$ over the whole recursion by Theorem 2.1. In the case of dual-pivot quicksort, the difference between $\mathcal{O}_2$ and $\mathcal{C}_2$ is $O(\log n)$, which also sums up to a difference of $O(n)$ over the whole recursion [Aumüller and Dietzfelbinger 2015]. It remains an open question to prove tighter bounds than $O(n^{11/12})$ in the general case.

5.3. A Fixed Strategy and its Algorithmic Variant

Now we turn to strategy $\mathcal{N}_k$: *Given $a_0, \dots, a_k$, the comparison tree $\lambda(v)$ used at node v is one that minimizes cost$^\lambda(a_0, \dots, a_k)$ over all comparison trees λ.*

Strategy $\mathcal{N}_k$ uses a fixed comparison tree for all classifications for given partition sizes, but it has to know these sizes in advance.

thm:n:k:optimal

THEOREM 5.3. *Strategy $\mathcal{N}_k$ is asymptotically optimal for each k.*

PROOF. According to Lemma 3.1 the average comparison count is determined up to lower order terms by the parameters $f^\lambda_{p_1,\dots,p_k}$, for each $\lambda \in \Lambda_k$. For each $p_1, \dots, p_k$, strategy $\mathcal{N}_k$ chooses the comparison tree which minimizes the average cost. By Lemma 3.1, this is optimal up to an $O(n^{1-\varepsilon})$ term. □

We will now describe how to implement strategy $\mathcal{N}_k$ by using sampling. Strategy $\mathcal{SP}_k$ works as follows: Let $\lambda_0 \in \Lambda_k$ be an arbitrary comparison tree. After the pivots are chosen, inspect the first $n^{3/4}$ elements and classify them using λ_0. Let $a'_0, \dots, a'_k$ denote the number of elements that belonged to $\mathrm{A}_0, \dots, \mathrm{A}_k$, respectively. Let λ be a comparison tree with minimal cost for $a'_0, \dots, a'_k$. Then classify each of the remaining elements by using λ.

thm:s:k:optimal

THEOREM 5.4. *Strategy $\mathcal{SP}_k$ is asymptotically optimal for each k.*

PROOF. Fix the k pivots $p_1, \dots, p_k$ and thus $a_0, \dots, a_k$. According to Lemma 3.1, the average comparison count $\mathrm{E}(P_n^{\mathcal{SP}_k} \mid p_1, \dots, p_k)$ can be calculated as follows:

$$\mathrm{E}(P_n^{\mathcal{SP}_k} \mid p_1, \dots, p_k) = \sum_{\lambda \in \Lambda_k} f^\lambda_{p_1,\dots,p_k} \cdot c^\lambda_{\text{avg}}(a_0, \dots, a_k) + O(n^{1-\varepsilon}).$$

Let λ^* be a comparison tree with minimal cost w. r. t. $a_0, \dots, a_k$. Let $a'_0, \dots, a'_k$ be the partition sizes after inspecting $n^{3/4}$ elements. Let λ be a comparison tree with minimal cost w. r. t. $a'_0, \dots, a'_k$. We call λ *good* if

$$c^\lambda_{\text{avg}}(a_0, \dots, a_k) - c^{\lambda^*}_{\text{avg}}(a_0, \dots, a_k) \leq \frac{2k}{n^{1/12}}, \text{ or equivalently}$$
$$\text{cost}^\lambda(a_0, \dots, a_k) - \text{cost}^{\lambda^*}(a_0, \dots, a_k) \leq 2kn^{11/12}, \tag{9}$$

eq:sp:k:2

otherwise we call λ *bad*. We define good_λ and bad_λ as the events that the sample yields a good and bad comparison tree, respectively.

We calculate:

$$\begin{aligned}\mathbf{E}(P_n^{\mathcal{SP}_k} \mid p_1,\ldots,p_k) &= \sum_{\substack{\lambda\in\Lambda_k\\ \lambda \text{ good}}} f^{\lambda}_{p_1,\ldots,p_k}\cdot c^{\lambda}_{\mathsf{avg}}(a_0,\ldots,a_k) + \\ &\qquad \sum_{\substack{\lambda\in\Lambda_k\\ \lambda \text{ bad}}} f^{\lambda}_{p_1,\ldots,p_k}\cdot c^{\lambda}_{\mathsf{avg}}(a_0,\ldots,a_k) + O(n^{1-\varepsilon}) \\ &\le n\cdot c^{\lambda^*}_{\mathsf{avg}}(a_0,\ldots,a_k) + \sum_{\substack{\lambda\in\Lambda_k\\ \lambda \text{ bad}}} f^{\lambda}_{p_1,\ldots,p_k}\cdot c^{\lambda}_{\mathsf{avg}}(a_0,\ldots,a_k) + O(n^{1-\varepsilon}) \\ &\le n\cdot c^{\lambda^*}_{\mathsf{avg}}(a_0,\ldots,a_k) + k\cdot \sum_{\substack{\lambda\in\Lambda_k\\ \lambda \text{ bad}}} f^{\lambda}_{p_1,\ldots,p_k} + O(n^{1-\varepsilon}). \qquad (10)\end{aligned}$$

eq:sp:k:1

Now we derive an upper bound for the second summand of (10). After the first $n^{3/4}$ classifications the algorithm will either use a good comparison tree or a bad comparison tree for the remaining classifications. The probability $\Pr(\mathsf{bad}_\lambda \mid p_1,\ldots,p_k)$ is the ratio of nodes on each level from $n^{3/4}$ to $n-k$ of the classification tree of nodes labeled with bad trees (in the sense of (9)). Summing over all levels, the second summand of (10) is thus at most $k\cdot n\cdot \Pr(\mathsf{bad}_\lambda \mid p_1,\ldots,p_k) + O(n^{3/4})$, where the latter summand collects error terms for the first $n^{3/4}$ steps.

LEMMA 5.5. *Conditioned on $p_1,\ldots,p_k$, good_λ occurs with very high probability.*

PROOF. For each $i\in\{0,\ldots,k\}$, let a'_i be the random variable that counts the number of elements from the sample that belong to group A_i. According to Lemma 2.2, with very high probability we have that $|a'_i - \mathbf{E}(a'_i)| \le n^{2/3}$, for each i with $0\le i\le k$. By the union bound, with very high probability there is no a'_i that deviates by more than $n^{2/3}$ from its expectation $n^{-1/4}\cdot a_i$. We will now show that if this happens then the event good_λ occurs. We obtain the following upper bound for an arbitrary comparison tree $\lambda'\in\Lambda_k$:

$$\begin{aligned}\mathsf{cost}^{\lambda'}(a'_0,\ldots,a'_k) &= \sum_{0\le i\le k} \mathsf{depth}_{\lambda'}(\mathrm{A}_i)\cdot a'_i \\ &\le \sum_{0\le i\le k} \mathsf{depth}_{\lambda'}(\mathrm{A}_i)\cdot n^{2/3} + n^{-1/4}\cdot \mathsf{cost}^{\lambda'}(a_0,\ldots,a_k) \\ &\le k^2 n^{2/3} + n^{-1/4}\cdot \mathsf{cost}^{\lambda'}(a_0,\ldots,a_k).\end{aligned}$$

Similarly, we get a corresponding lower bound. Thus, for each comparison tree $\lambda'\in\Lambda_k$ it holds that

$$\frac{\mathsf{cost}^{\lambda'}(a_0,\ldots,a_k)}{n^{1/4}} - k^2 n^{2/3} \le \mathsf{cost}^{\lambda'}(a'_0,\ldots,a'_k) \le \frac{\mathsf{cost}^{\lambda'}(a_0,\ldots,a_k)}{n^{1/4}} + k^2 n^{2/3},$$

and we get the following bound:

$$\begin{aligned}&\mathsf{cost}^{\lambda}(a_0,\ldots,a_k) - \mathsf{cost}^{\lambda^*}(a_0,\ldots,a_k) \\ &\qquad \le n^{1/4}\big(\mathsf{cost}^{\lambda}(a'_0,\ldots,a'_k) - \mathsf{cost}^{\lambda^*}(a'_0,\ldots,a'_k)\big) + 2n^{1/4}\cdot k^2\cdot n^{2/3} \\ &\qquad \le 2k^2\cdot n^{11/12}.\end{aligned}$$

(The last inequality follows because λ has minimal cost w. r. t. $a'_0,\ldots,a'_k$.) Hence, λ is good. □

Thus, the average comparison count of $\mathcal{SP}_k$ is at most a summand of $O(n^{1-\varepsilon})$ larger than the average comparison count of $\mathcal{N}_k$. This implies that $\mathcal{SP}_k$ is asymptotically optimal as well. □

Since the number of comparison trees in Λ_k is exponentially large in k, one might want to restrict the set of used comparison trees to some subset $\Lambda'_k \subseteq \Lambda_k$. We remark here that our strategies are optimal w. r. t. any chosen subset of comparison trees as well.

omparison:median:of:k

6. THE OPTIMAL AVERAGE COMPARISON COUNT OF k-PIVOT QUICKSORT

In this section we use the theory developed so far to discuss the optimal average comparison count of k-pivot quicksort. We compare the result to the well known median-of-k strategy of classical quicksort [van Emden 1970].

By Lemma 3.1 and Theorem 5.3, the minimal partitioning cost for k-pivot quicksort (up to lower order terms) is

$$\frac{1}{\binom{n}{k}} \sum_{a_0+\dots+a_k=n-k} \min\Big\{\operatorname{cost}^\lambda(a_0,\dots,a_k) \mid \lambda \in \Lambda_k\Big\} + O(n^{1-\varepsilon}). \tag{11}$$

eq:optimal:cost:form

Then applying Theorem 2.1 gives the minimal average comparison count for k-pivot quicksort.

Unfortunately, we were not able to solve (11) for $k \geq 4$. (Already the solution for $k = 3$ as stated in Section 4 required a lot of manual tweaking before using Maple®.) This remains an open question. We resorted to experiments. As noticed in [Aumüller and Dietzfelbinger 2013], estimating the total average comparison count by sorting inputs does not allow us to estimate the leading term of the average comparison count correctly, because the $O(n)$ term in (1) has a big influence on the average comparison count for real-world input lengths. We used the following approach instead: For $n = 50 \cdot 10^6$, we generated $10\,000$ random permutations of $\{1,\dots,n\}$ and ran strategy $\mathcal{O}_k$ for each input, i. e., only classified the input.[3] For the average partitioning cost measured in these experiments, we then applied (1) to derive the leading factor of the total average comparison count. Table I shows the results from these experiments for $k \in \{2,\dots,9\}$. Note that the results for $k \in \{2,3\}$ are almost identical to the exact theoretical results. Additionally, the table shows the theoretical results known for classical quicksort using the median-of-k strategy [van Emden 1970; Hennequin 1991]. Interestingly, from Table I we see that—based on our experimental data for k-pivot quicksort—the median-of-k strategy has—starting from $k = 7$—a slightly lower average comparison count than the (rather complicated) optimal partitioning methods for k-pivot quicksort.

We close our study of comparison-optimal k-pivot quicksort with one remark about the practical influence of optimal k-pivot partitioning. For $k \geq 2$, neither strategy $\mathcal{C}_k$ nor $\mathcal{SP}_k$ can compete in running time even with classical quicksort when the optimal comparison tree is computed by the dynamic programming algorithm referenced in Section 5.1. For $k = 2$, experiments in [Aumüller and Dietzfelbinger 2013] showed that these approaches cannot compete with Yaroslavskiy's dual-pivot algorithm, either, even when we bypass the dynamic programming algorithm. (For $k = 2$ there exist only two comparisons trees.)

sec:assignments

7. REARRANGING ELEMENTS

With this section, we change our viewpoint on multi-pivot quicksort in two respects: we consider cost measures different than comparisons and focus on one particularly interesting algorithm for the "rearrangement problem". The goal now is to find other

[3] Experiments with other input sizes gave exactly the same results.

tab:optimal:cost:k:pivot:quicksort

Table I. Optimal average comparison count for k-pivot quicksort for $k \in \{2, \dots, 9\}$. Note that the values for $k \geq 4$ are based on experiments. For odd k, we also include the average comparison count of quicksort with the median-of-k strategy. (The numbers for the median-of-k variant can be found in [van Emden 1970] or [Hennequin 1991].)

Pivot Number k	**opt. k-pivot**	**median-of-k**
2	$1.800n \ln n$	—
3	$1.705n \ln n$	$1.714n \ln n$
4	$1.650n \ln n$	—
5	$1.610n \ln n$	$1.622n \ln n$
6	$1.590n \ln n$	—
7	$1.577n \ln n$	$1.576n \ln n$
8	$1.564n \ln n$	—
9	$1.555n \ln n$	$1.549n \ln n$

cost measures which show differences in multi-pivot quicksort algorithms with respect to running time in practice.

7.1. Which Factors are Relevant for Running Time?

Let us first reflect on the influence of key comparisons to running time. From a running time perspective it seems unintuitive that comparisons are the crucial factor with regard to running time, especially when key comparisons are cheap, e. g., when comparing 32-bit integers. However, while a comparison is often cheap, mispredicting the destination that a branch takes, i. e., the outcome of the comparison, may incur a significant penalty in running time, because the CPU wasted work on executing instructions on the wrongly predicted branch. One famous example for the effect of branch prediction is [Kaligosi and Sanders 2006] in which quicksort is made faster by choosing a skewed pivot due to pipelining effects on a certain CPU. In very recent work, Martínez et al. [2015] considered differences in branch misses between classical quicksort and Yaroslavskiy's algorithm, but found no crucial differences. They concluded that the advantages in running time of the dual-pivot approach are not due to differences in branch prediction.

Traditionally, the *cost of moving elements around* is also considered as a cost measure of sorting algorithms. This cost is usually expressed as the number of *swap* operations or the number of *assignments* needed to sort the input. [Kushagra et al. 2014] take a different approach and concentrate on the I/O performance of quicksort variants with respect to their *cache behavior*. The I/O performance is often a bottleneck of an algorithm because an access to main memory in modern computers can be slower than executing a few hundred simple CPU instructions. Caches speed these accesses up, but their influence seems difficult to analyze. Let us exemplify the influence of caches on running time. First, the cache structure of modern CPU's is usually hierarchical. For example, the Intel i7 that we used in our experiments has three data caches: There is a very small L1 cache (32KB of data) and a slightly larger L2 cache (256KB of data) very close to the processor. Each CPU core has its own L1 and L2 cache. They are both 8-way associative, i. e., a memory segment can be stored at eight different cache lines. Shared among cores is a rather big L3 cache that can hold 8MB of data and is 16-way associative. Caches greatly influence running time. While a lookup in main memory costs many CPU cycles (≈ 140 cycles on the Intel i7 used in our experiments), a cache access is very cheap and costs about 4, 11, and 25 cycles for a hit in L1, L2, and L3 cache, respectively [Levinthal 2009]. Also, modern CPU's use *prefetching* to load memory segments into

cache before they are accessed. Usually, there exist different prefetchers for different caches, and there exist different strategies to prefetch data, e. g., "load two adjacent cache lines", or "load memory segments based on predictions by monitoring data flow".

From a theoretical point of view, much research has been conducted to study algorithms with respect to their cache behavior, see, e. g., the survey paper of Rahman [2002]. (We recommend this paper as an excellent introduction to the topic of caches.)

In [Kushagra et al. 2014] a fast three-pivot algorithm was described. They analyzed its cache behavior and compared it to classical quicksort and Yaroslavsiky's dual-pivot quicksort algorithm using the approach of [LaMarca and Ladner 1999]. Their results gave reason to believe that the improvements of multi-pivot quicksort algorithms with respect to running times result from their better cache behavior. They also reported from experiments with a seven-pivot algorithm, which ran more slowly than their three-pivot algorithm. Very recently, [Nebel et al. 2015] gave a more detailed analysis of the cache behavior of Yaroslavskiy's algorithm also with respect to different sampling strategies. An important contribution of [Nebel et al. 2015] is the distinction of a theoretical measure *scanned elements* (basically the number of times a memory cell is inspected during sorting) and the usage of this cost measures to predict cache behavior.

In this section we discuss how the considerations of [Kushagra et al. 2014; Nebel et al. 2015] generalize to the case of using more than three pivots. In connection with the running time experiments from Section 9, this allows us to make more accurate predictions than [Kushagra et al. 2014] about the influence of cache behavior on running time. One result of this study will be that it is not surprising that their seven-pivot approach is slower, because it has worse cache behavior than three- or five-pivot quicksort algorithms using a specific partitioning strategy.

We will start by specifying the problem setting, and subsequently introduce a generalized partitioning algorithm for k pivots. This algorithm is the generalization of the partitioning methods used in classical quicksort, Yaroslavskiy's algorithm, and the three-pivot quicksort algorithm of [Kushagra et al. 2014]. This strategy will be evaluated for different values of k with respect to different memory-related cost measures which will be introduced later. It will turn out that these theoretical cost measures allow us to give detailed recommendations under which circumstances a multi-pivot quicksort approach has advantages over classical quicksort. At the end of this section, we will compare this algorithm to other algorithms from the literature that can be used as partitioning algorithms.

cost:measures:problem

7.2. The Rearrangement Problem

With regard to counting key comparisons we defined the classification problem to abstract from the situation that a multi-pivot quicksort algorithm has to move elements around to produce the partition. Here, we assume that for each element its groups is known and we are only interested in moving elements around to produce the partition. This motivates us to consider the *rearrangement problem for k pivots*: Given a sequence of length $n - k$ with entries having labels from the set $\{\mathrm{A}_0, \ldots, \mathrm{A}_k\}$ of group names, the task is to rearrange the entries with respect to their labels into ascending order, where $\mathrm{A}_i < \mathrm{A}_{i+1}$ for $i \in \{0, \ldots, k-1\}$. Note that any classification strategy can be used to find out element groups. We assume that the input resides in an array $A[1..n]$ where the k first cells hold the pivots.[4] For $k = 2$, this problem is known under the name *Dutch national flag problem*, proposed by Dijkstra [Dijkstra 1976]. For $k > 2$, the problem was considered in the paper of McIlroy et al. [1993], who devised an algorithm called

[4] We shall disregard the pivots in the description of the problem. In a final step the k pivots have to be moved into the correct positions between group segments. This is possible by moving not more than k^2 elements around using k rotate operations, as introduced below.

"American flag sort" to solve the rearrangement problem for $k > 2$. We will discuss the applicability of these algorithms at the end of this section. Our goal is to analyze algorithms for this problem with respect to different cost measures, e. g., the number of array cells that are inspected during rearranging the input, or the number of times the algorithm writes to array cells in the process. We start by introducing an algorithm for the rearrangement problem that generalizes the algorithmic ideas behind rearranging elements in classical quicksort, Yaroslavskiy's dual-pivot quicksort [Nebel et al. 2015], and the three-pivot algorithm of [Kushagra et al. 2014].

t:measures:algorithms

7.3. The Algorithm

To capture the cost of rearranging the elements, in the analysis of sorting algorithms one traditionally uses the "swap"-operation, which exchanges two elements. The cost of rearranging is then just the number of swap operations performed during the sorting process. In the case that one uses two or more pivots, we we will see that it is beneficial to generalize this operation. We define the operation $\texttt{rotate}(i_1, \ldots, i_\ell)$ as follows:

$$\text{tmp} \leftarrow A[i_1]; A[i_1] \leftarrow A[i_2]; A[i_2] \leftarrow A[i_3]; \ldots; A[i_{\ell-1}] \leftarrow A[i_\ell]; A[i_\ell] \leftarrow \text{tmp}.$$

The operation `rotate` performs a cyclic shift of the elements by one position. A $\texttt{swap}(A[i_1], A[i_2])$ is a $\texttt{rotate}(i_1, i_2)$. A $\texttt{rotate}(i_1, \ldots, i_\ell)$ operation makes exactly $\ell + 1$ assignments and inspects and writes into ℓ array cells.

For each $k \geq 1$ we consider an algorithm *Exchange*$_k$. Pseudocode of this algorithm is given in Algorithm 1. The basic idea is similar to classical quicksort: Two pointers[5] scan the array. One pointer scans the array from left to right; another pointer scans the array from right to left, exchanging misplaced elements on the way. Formally, the algorithm uses two pointers `i` and `j`. At the beginning, `i` points to the element in $A[k+1]$ and `j` points to the element in $A[n]$. We set $m = \lceil \frac{k+1}{2} \rceil$. The algorithm makes sure that all elements to the left of pointer `i` belong to groups $\mathrm{A}_0, \ldots, \mathrm{A}_{m-1}$ (and are arranged in this order), i. e., m is the number of groups left of pointer `i`. Also, all elements to the right of pointer `j` belong to groups $\mathrm{A}_m, \ldots, \mathrm{A}_k$, arranged in this order. To do so, Algorithm 1 uses $k-1$ additional "border pointers" $\texttt{b}_1, \ldots, \texttt{b}_{k-1}$. For $i < m$, the algorithm makes sure that at each point in time, pointer $\texttt{b}_i$ points to the leftmost element to the left of pointer `i` which belongs to group $\mathrm{A}_{i'}$, where $i' \geq i$. Analogously, for $j \geq m$, the algorithm makes sure that pointer $\texttt{b}_j$ points to the rightmost element to the right of pointer `j` which belongs to group $\mathrm{A}_{j'}$ with $j' \leq j$, see Figure 4. As long as pointers `i` and `j` have not crossed yet, the algorithm increments pointer `i` until `i` points to an element that belongs to a group A_p with $p \geq m$. For each element x along the way that belongs to a group $\mathrm{A}_{p'}$ with $p' < m-1$, it moves x to the place to which $\texttt{b}_{p'+1}$ points, using a rotate operation to make space to accommodate the element, see Figure 5 and Lines 7–11 in Algorithm 1. Pointers $\texttt{b}_{p'+1}, \ldots, \texttt{b}_{m-1}$ are incremented afterwards. Similarly, the algorithm decrements pointer `j` until it points to an element that belongs to a group A_q with $q < m$, moving elements from $\mathrm{A}_{m+1}, \ldots, \mathrm{A}_k$ along the way in a similar fashion, see Figure 6 and Line 12–16 in Algorithm 1. If now $\texttt{i} < \texttt{j}$, a single rotate operation suffices to move the elements referenced by `i` and `j` to a (temporarily) correct position, see Figure 7 and Line 17–20 in Algorithm 1. Note that any classification strategy can be used in an "online fashion" to find out element groups in Algorithm 1.

Figure 4 shows the idea of the algorithm for $k = 6$; Figures 5–7 show the different rotations being made by Algorithm 1 in lines 9, 14, and 18.

[5] Note that our pointers are actually variables that hold an array index.

algo:exchange:k

Algorithm 1 Move elements by rotations to produce a partition

procedure *Exchange*$_k(A[1..n])$

```
1: i ← k + 1; j ← n;
2: m ← ⌈(k+1)/2⌉;
3: b_1, ..., b_{m-1} ← i;
4: b_m, ..., b_{k-1} ← j;
5: p, q ← −1;        ▷ p and q hold the group indices of the elements indexed by i and j.
6: while i < j do
7:     while A[i] belongs to group A_p with p < m do
8:         if p < m − 1 then
9:             rotate(i,b_{m−1}, ..., b_{p+1});
10:            b_{p+1}++; ...; b_{m−1}++;
11:        i++;
12:    while A[j] belongs to group A_q with q ≥ m do
13:        if q ≥ m + 1 then
14:            rotate(j,b_m, ..., b_{q−1});
15:            b_{q−1}--; ...; b_m--;
16:        j--;
17:    if i < j then
18:        rotate(i, b_{m−1}, ..., b_{q+1}, j, b_m, ..., b_{p−1});
19:        i++; b_{q+1}++; ...; b_{m−1}++;
20:        j--; b_m--; ...; b_{p−1}--;
```

7.4. Cost Measures and Assumptions of the Analysis

In the following we consider three cost measures as cost for rearranging the input using Algorithm 1. The first two cost measures aim to describe the memory behavior of Algorithm 1. The first measure counts how often each array cell is accessed during rearranging the input, which in practice gives a good approximation on the time the CPU has to wait for memory, even when the data is in cache. We will show later that this theoretical cost measures allows us to describe practical cost measures like the average number of cache misses accurately. The second cost measure counts how often the algorithm writes into an array cell. The last cost measure is more classical and counts how many assignments the algorithm makes. It will be interesting to see that while these cost measures appear to be similar, only the first one will correctly reflect advantages of a multi-pivot quicksort approach in empirical running time. The first cost measure was also considered for Yaroslavskiy's algorithm in Nebel et al. [2015].

Keine Empirical Running Time Betrachtungen dort.

Scanned Elements. Assume that a pointer `l` is initialized with value l_s. Let l_e be the value in `l` after the algorithm finished rearranging the input. Then we define $\text{cost}(\texttt{l}) = |l_s - l_e|$, i. e., the number of array cells inspected by pointer `l`. (Note that a cell is accessed only once per pointer, all pointers move by increments or decrements of 1, and $A[l_e]$ is not inspected.) Let the variable P^{se} be the *number of scanned elements* of Algorithm 1. It is the sum of the costs of pointers $\texttt{i}, \texttt{j}, \texttt{b}_1, \ldots, \texttt{b}_{k-1}$. From an empirical point of view this cost measure gives a lower bound on the number of clock cycles the CPU spends waiting for memory. It can also be used to predict the cache behavior of Algorithm 1. We will see that it gives good estimates for the cache misses in L1 cache which we observed in our experiments. This has also been observed in the special case of Yaroslavskiy's algorithm in [Nebel et al. 2015].

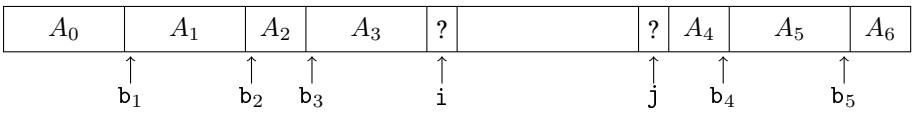

Fig. 4. General memory layout of Algorithm 1 for $k = 6$. Two pointers i and j are used to scan the array from left to right and right to left, respectively. Pointers $b_1, \ldots, b_{k-1}$ are used to point to the start (resp. end) of segments.

fig:partition:k

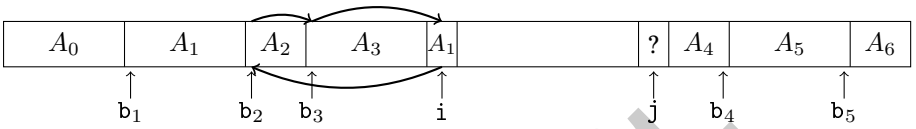

Fig. 5. The rotate operation in Line 9 of Algorithm 1. An element that belongs to group A_1 is moved into its respective segment. Pointers i, b_2, b_3 are increased by 1 afterwards.

fig:partition:k:rotat

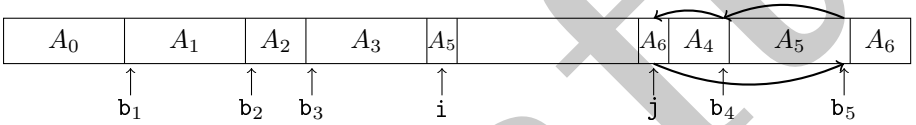

Fig. 6. The rotate operation in Line 14 of Algorithm 1. An element that belongs to group A_6 is moved into its respective segment. Pointers j, b_4, b_5 are decreased by 1 afterwards.

fig:partition:k:rotat

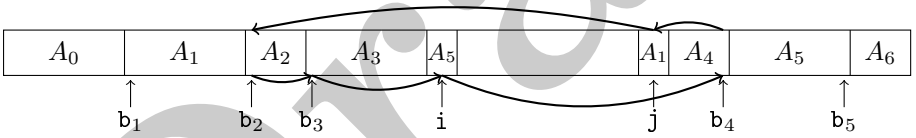

Fig. 7. Example for the rotate operation in Line 18 of Algorithm 1. The element found at i is moved into its specific segment. Subsequently, the element found at j is moved into its specific segment.

fig:partition:k:rotat

Write Accesses. Each rotate operation of ℓ elements of Algorithm 1 writes into exactly ℓ array cells. When we assign a value to an array, we call the access to this array cell a *write access*. Let the variable P^{wa} be the *number of write accesses* (over all rotate operations) of write accesses into array cells.

Assignments. Each rotate operation of ℓ elements of Algorithm 1 makes exactly $\ell + 1$ assignments. Let the variable P^{as} be the *number of assignments* over all rotate operations. Since each swap operation consists of three assignments, this is the most classical cost measure with respect to the three cost measures introduced above for the analysis of quicksort.

Setup of the Analysis. In the following we want to obtain the leading term for the average number of scannend elements, write accesses, and assignments, both for partitioning and over the whole sorting process. The input is again assumed to be a random permutation of the set $\{1, \ldots, n\}$ which resides in an array $A[1..n]$. Fix an integer $k \geq 1$. The first k elements are chosen as pivots. Then we can think of the input consisting of $n - k$ elements having labels from $A_0, \ldots, A_k$, and our goal is to rearrange the input. (In terms of multi-pivot quicksort, our goal is to obtain a partition of the input, as depicted in Figure 1 on Page 2. However, here determining to which of the groups $A_0, \ldots, A_k$ element $A[i]$ belongs is for free.) We are interested in the cost of the rearrangement process and the total sorting cost in the cost measures introduced above.

From Partitioning Cost to Sorting Cost. Let P_n denote the partitioning cost that the algorithm incurs in the first partitioning/rearrangement step. Let the random variable C_n count the sorting cost (over the whole recursion) of sorting an input of length n in the respective cost measure. As before, we get the recurrence:

$$\mathbf{E}(C_n) = \mathbf{E}(P_n) + \frac{1}{\binom{n}{k}} \sum_{a_0+\cdots+a_k=n-k} (\mathbf{E}(C_{a_0}) + \cdots + \mathbf{E}(C_{a_k})).$$

Again, this recurrence has the form of (2), so we may apply (3) for linear partitioning cost. Thus, from now on we focus on a single partitioning step.

7.5. Analysis

Our goal in this section is to prove the following theorem. A discussion of this result will be given in the next section.

THEOREM 7.1. *Let $k \geq 1$ be the number of pivots and $m = \lceil \frac{k+1}{2} \rceil$. Then for Algorithm 1, we have that*

$$\mathbf{E}(\mathbf{P}_n^{\text{se}}) = \begin{cases} \dfrac{m+1}{2} \cdot n + O(1), & \text{for odd } k, \\ \dfrac{m^2}{2m-1} \cdot n + O(1), & \text{for even } k, \end{cases} \tag{12}$$

eq:memory:accesses:e

$$\mathbf{E}(\mathbf{P}_n^{\text{wa}}) = \begin{cases} \dfrac{2m^3+3m^2-m-2}{2m(2m+1)} \cdot n + O(1), & \text{for odd } k, \\ \dfrac{2m^3-2m-1}{2m(2m-1)} \cdot n + O(1), & \text{for even } k, \end{cases} \tag{13}$$

eq:write:accesses:ex

$$\mathbf{E}(\mathbf{P}_n^{\text{as}}) = \begin{cases} \dfrac{2m^3+6m^2-m-4}{2m(2m+1)} \cdot n + O(1), & \text{for odd } k, \\ \dfrac{2m^3+3m^2-5m-2}{2m(2m-1)} \cdot n + O(1), & \text{for even } k, \end{cases} \tag{14}$$

eq:assignments:excha

thm:partition:cost

From this theorem, we get the leading term of the total sorting cost in the respective cost measure by applying (3).

measures:cache:misses

Scanned Elements. We will first study how many elements are scanned by the pointers used in Algorithm 1 when sorting an input.

Let the pivots and thus $a_0, \ldots, a_k$ be fixed. The pointers `i` and `j` together scan the whole array, and thus inspect $n-k$ array cells. When Algorithm 1 terminates, $\mathtt{b}_1$ points to $A[k+a_0+1]$, having visited exactly a_0 array cells. An analogous statement can be made for the pointers $\mathtt{b}_2, \ldots, \mathtt{b}_{k-1}$. On average, we have $(n-k)/(k+1)$ elements of each group $A_0, \ldots, A_k$, so $\mathtt{b}_1$ and $\mathtt{b}_{k-1}$ each visit $(n-k)/(k+1)$ array cells on average, $\mathtt{b}_2$ and $\mathtt{b}_{k-2}$ each visit $2(n-k)/(k+1)$ array cells, and so on.

For the average number of scanned elements in a partitioning step we consequently get

$$\mathrm{E}(\mathrm{P}_n^{\mathrm{se}}) = \begin{cases} 2 \cdot \sum_{i=1}^{\lceil k/2 \rceil} \frac{i \cdot (n-k)}{k+1}, & \text{for odd } k, \\ 2 \cdot \sum_{i=1}^{k/2} \frac{i \cdot (n-k)}{k+1} + \frac{k/2+1}{k+1} \cdot (n-k), & \text{for even } k, \end{cases} \tag{15}$$

eq:memory:accesses:1

and a simple calculation shows

$$\mathrm{E}(\mathrm{P}_n^{\mathrm{se}}) = \begin{cases} \left(\frac{m+1}{2}\right) \cdot (n-k), & \text{for odd } k, \\ \left(\frac{m^2}{2m-1}\right) \cdot (n-k), & \text{for even } k. \end{cases} \tag{16}$$

eq:memory:accesses:ex

Write Accesses. We now focus on the average number of write accesses. First we observe that a rotate operation involving ℓ elements in Algorithm 1 makes exactly ℓ element scans and ℓ write accesses. So, the only difference between element scans and write accesses is that whenever pointer i finds an A_{m-1}-element in Line 7 or pointer j finds an A_m-element in Line 12, the element is scanned but no write access takes place. Let $C_{\mathtt{i},m-1}$ be the random variable that counts the number of A_{m-1}-elements found in Line 7, and let $C_{\mathtt{j},m}$ be the random variable that counts the number of A_m-elements found in Line 12 of Algorithm 1.

Thus, we know that

$$\mathrm{E}(\mathrm{P}_n^{\mathrm{wa}}) = \mathrm{E}(\mathrm{P}_n^{\mathrm{se}}) - \mathrm{E}(C_{\mathtt{i},m-1}) - \mathrm{E}(C_{\mathtt{j},m}). \tag{17}$$

eq:from:scanned:eleme

LEMMA 7.2. *Let k be the number of pivots and let $m = \lceil \frac{k+1}{2} \rceil$. Then*

$$\mathrm{E}(C_{\mathtt{i},m-1}) + \mathrm{E}(C_{\mathtt{j},m}) = \begin{cases} \frac{m+1}{m(2m+1)} n + O(1), & \textit{for } k \textit{ odd}, \\ \frac{2m+1}{2m(2m-1)} n + O(1), & \textit{for } k \textit{ even}. \end{cases}$$

ectly:placed:elements

PROOF. We start by obtaining bounds on $\mathrm{E}(C_{\mathtt{i},m-1})$ and $\mathrm{E}(C_{\mathtt{j},m})$ when k is even. The calculations for the case when k is odd are simpler because of symmetry. In the calculations, we will consider the two events that the groups $\mathrm{A}_0, \ldots, \mathrm{A}_{m-1}$ have L elements in total, for $0 \le L \le n-k$, and that group A_{m-1} has K elements, for $0 \le K \le L$. If the group sizes are as above, then the expected number of A_{m-1} elements scanned by pointer i is $L \cdot K/(n-k)$. We first observe that

$$\begin{aligned} \mathrm{E}(C_{\mathtt{i},m-1}) &= \sum_{L=0}^{n} \sum_{K=0}^{L} \Pr(a_0 + \cdots + a_{m-1} = L \wedge a_{m-1} = K) \cdot L \cdot \frac{K}{n-k} \\ &\overset{(*)}{=} \frac{1}{n \cdot m} \sum_{L=0}^{n} \Pr(a_0 + \cdots + a_{m-1} = L) \cdot L^2 + O(1) \\ &= \frac{1}{n \cdot m} \sum_{L=1}^{n} \frac{\binom{L-1}{m-1}\binom{n-L}{m-2}}{\binom{n}{2(m-1)}} \cdot L^2 + O(1), \end{aligned} \tag{18}$$

eq:proof:write:access

where $(*)$ follows by noticing that $\sum_{K=0}^{L} \Pr(a_{m-1} = K \mid a_0 + \cdots + a_{m-1} = L) \cdot K$ is just the expected size of the group A_{m-1} given that the first m groups have exactly L elements, which is L/m. We calculate the sum of binomial coefficients as in (18) for a more general situation:

CLAIM 7.3. *Let ℓ_1 and ℓ_2 be arbitrary integers. Then we have*

$$\text{(i)} \sum_{L=1}^{n} \frac{\binom{L-1}{\ell_1}\binom{n-L}{\ell_2}}{\binom{n}{\ell_1+\ell_2+1}} \cdot L^2 = \frac{(\ell_1+1)(\ell_1+2)\cdot(n+2)(n+1)}{(\ell_1+\ell_2+2)(\ell_1+\ell_2+3)} - \frac{(\ell_1+1)\cdot(n+1)}{(\ell_1+\ell_2+2)}.$$

$$\text{(ii)} \sum_{L=1}^{n} \frac{\binom{L-1}{\ell_1}\binom{n-L}{\ell_2}}{\binom{n}{\ell_1+\ell_2+1}} \cdot (n-L)^2 = \frac{(\ell_2+1)(\ell_2+2)\cdot(n+2)(n+1)}{(\ell_1+\ell_2+2)(\ell_1+\ell_2+3)} - \frac{3(\ell_2+1)\cdot(n+1)}{\ell_1+\ell_2+2} + 1.$$

binomial:coefficients

PROOF. We denote by $n^{\underline{k}}$ the k-th falling factorial of n, i. e., $n(n-1)\cdots(n-k+1)$. Using known identities for sums of binomial coefficients, we may calculate

$$\begin{aligned}
&\sum_{L=1}^{n} \frac{\binom{L-1}{\ell_1}\binom{n-L}{\ell_2}}{\binom{n}{\ell_1+\ell_2+1}} \cdot L^2 \\
&= \frac{1}{\binom{n}{\ell_1+\ell_2+1}} \sum_{L=1}^{n} \left((\ell_1+1)(\ell_1+2)\binom{L+1}{\ell_1+2}\binom{n-L}{\ell_2} - L\binom{L-1}{\ell_1}\binom{n-L}{\ell_2} \right) \\
&= \frac{1}{\binom{n}{\ell_1+\ell_2+1}} \sum_{L=1}^{n} \left((\ell_1+1)(\ell_1+2)\binom{L+1}{\ell_1+2}\binom{n-L}{\ell_2} - (\ell_1+1)\binom{L}{\ell_1+1}\binom{n-L}{\ell_2} \right) \\
&\overset{(*)}{=} \frac{1}{\binom{n}{\ell_1+\ell_2+1}} \left((\ell_1+1)(\ell_1+2)\binom{n+2}{\ell_1+\ell_2+3} - (\ell_1+1)\binom{n+1}{\ell_1+\ell_2+2} \right) \\
&= \frac{(\ell_1+1)(\ell_1+2)(n+2)^{\underline{\ell_1+\ell_2+3}}(\ell_1+\ell_2+1)!}{(\ell_1+\ell_2+3)!\cdot n^{\underline{\ell_1+\ell_2+1}}} - \frac{(\ell_1+1)(n+1)^{\underline{\ell_1+\ell_2+2}}(\ell_1+\ell_2+1)!}{(\ell_1+\ell_2+2)!\cdot n^{\underline{\ell_1+\ell_2+1}}} \\
&= \frac{(\ell_1+1)(\ell_1+2)(n+2)(n+1)}{(\ell_1+\ell_2+2)(\ell_1+\ell_2+3)} - \frac{(\ell_1+1)(n+1)}{(\ell_1+\ell_2+2)},
\end{aligned}$$

where $(*)$ follows by using the identity [Graham et al. 1994, (5.26)]. The calculations for (ii) are analogous by using an index transformation $K = n - L$. □

Using the claim, we continue from (18) as follows:

$$\mathrm{E}(C_{\mathtt{i},m-1}) = \frac{m(m+1)\cdot(n+1)(n+2)}{nm\cdot(2m-1)2m} + O(1) = \frac{m+1}{2m(2m-1)}n + O(1). \tag{19}$$

eq:proof:write:acces

By similar arguments, we obtain

$$\begin{aligned}
\mathrm{E}(C_{\mathtt{j},m}) &= \frac{1}{n\cdot(m-1)} \sum_{L=0}^{n} \frac{\binom{L}{m-1}\binom{n-L}{m-2}}{\binom{n}{2(m-1)}} \cdot (n-L)^2 + O(1) \\
&= \frac{(m-1)m}{2nm(m-1)(2m-1)}n^2 + O(1) = \frac{1}{2(2m-1)}n + O(1). \qquad (20)
\end{aligned}$$

eq:proof:write:acces

Thus, in the asymmetric case it holds that $\mathrm{E}(C_{\mathtt{i},m-1}) + \mathrm{E}(C_{\mathtt{j},m}) = \frac{2m+1}{2m(2m-1)}n + O(1)$.

Applying Lemma 7.2 to (16) and (17) and simplifying gives us the value from Theorem 7.1.

Assignments. To count the total number of assignments, we first observe that each rotate operation that involves ℓ elements makes ℓ write accesses and $\ell + 1$ assignments. Thus, the total number of assignment is just the sum of the number of write accesses and the number of rotate operations. So we observe

$$\mathrm{E}(\mathrm{P}_n^{\mathrm{as}}) = \mathrm{E}(\mathrm{P}_n^{\mathrm{wa}}) + \mathrm{E}(\#\text{rotate operations}). \tag{21}$$

eq:from:write:accesse

LEMMA 7.4. *Let k be the number of pivots and $m = \lceil \frac{k+1}{2} \rceil$. Then it holds that*

$$\mathrm{E}(\#\text{rotate operations}) = \begin{cases} \frac{3m^2-2}{2m(2m+1)}, & \text{for odd } k, \\ \frac{3m^2-3m-1}{2m(2m-1)}, & \text{for even } k. \end{cases}$$

avg:rotate:operations

PROOF. The number of rotate operations is counted as follows. For each non-A_{m-1} element that is scanned by pointer `i`, a rotate operation is invoked (Line 9 and Line 18 in Algorithm 1). In addition, each $A_{m'}$ element with $m' > m$ scanned by pointer `j` invokes a rotate operation (Line 14 in Algorithm 1). So, the number of rotate operations is the sum of these two quantities. Again, we focus on the case that k is even. Let $C_{\mathtt{i},<m-1}$ be the number of $A_{m'}$ elements with $m' < m-1$ scanned by pointer `i`. Define $C_{\mathtt{i},>m-1}$ and $C_{\mathtt{j},>m}$ analogously. By symmetry (cf. (18)) we have that

$$\mathrm{E}(C_{\mathtt{i},<m-1}) = (m-1) \cdot \mathrm{E}(C_{\mathtt{i},m-1}) = \frac{(m-1)(m+1)}{2m(2m-1)} n + O(1),$$

see (19). Furthermore, since we expect that pointer `i` scans $\frac{m}{2m-1}(n-k)$ elements, we know that

$$\mathrm{E}(C_{\mathtt{i},>m-1}) = \frac{m}{2m-1}(n-k) - m \cdot \mathrm{E}(C_{\mathtt{i},m-1}) = \left(\frac{m}{2m-1} - \frac{m+1}{2(2m-1)} \right) n + O(1).$$

Finally, again by symmetry we obtain

$$\mathrm{E}(C_{\mathtt{j},>m}) = (m-2) \cdot \mathrm{E}(C_{\mathtt{j},m}) = \frac{m-2}{2(2m-1)} n + O(1).$$

For even k the result now follows by adding these three values. For odd k, we only have to adjust that we expect that pointer `i` scans $(n-k)/2$ elements, and that there are $m-1$ groups $\mathrm{A}_{m+1}, \ldots, \mathrm{A}_k$ when calculating $\mathrm{E}(C_{\mathtt{j},>m})$. □

Applying Lemma 7.4 to (21) and simplifying gives the value from Theorem 7.1.

7.6. Discussion and Empirical Validation

Using the formulae developed in the previous subsection we calculated the average number of scanned elements, write accesses, and assignments in partitioning and in sorting for $k \in \{1, \ldots, 9, 15, 31\}$ using Theorem 7.1 and (3). Table II shows the results of these calculations. Next, we will discuss our findings.

Interestingly, Algorithm 1 improves over classical quicksort when using more than one pivot with regard to scanned elements. A 3-pivot quicksort algorithm, using this partitioning algorithm, has lower cost than classical and dual-pivot quicksort. Moreover, the average number of scanned elements is minimized by the 5-pivot partitioning algorithm. However, the difference to the 3-pivot algorithm is small. Using more than 5 pivots increases the average number of scanned elements. With respect to write accesses and assignments, we see a different picture. In both cost measures, the average sorting cost rapidly increases from classical quicksort to quicksort variants with at least two pivots. For a growing number of pivots, it slowly increases. In conclusion, Algorithm 1

tab:partition:cost

Table II. Average number of scanned elements ($\mathrm{E}(\mathrm{P}_n^{\mathrm{se}})$), write accesses ($\mathrm{E}(\mathrm{P}_n^{\mathrm{wa}})$), and assignments ($\mathrm{E}(\mathrm{P}_n^{\mathrm{as}})$) for partitioning together with the total sorting cost for sorting an input of length n disregarding lower order terms and factors rounded using k pivots.

k	$\mathrm{E}(\mathrm{P}_n^{\mathrm{se}})$	$\mathrm{E}(\mathrm{C}_n^{\mathrm{se}})$	$\mathrm{E}(\mathrm{P}_n^{\mathrm{wa}})$	$\mathrm{E}(\mathrm{C}_n^{\mathrm{wa}})$	$\mathrm{E}(\mathrm{P}_n^{\mathrm{as}})$	$\mathrm{E}(\mathrm{C}_n^{\mathrm{as}})$
1	$1.000n$	$2.000n \ln n$	$0.333n$	$0.667n \ln n$	$0.500n$	$1.000n \ln n$
2	$1.333n$	$1.600n \ln n$	$0.917n$	$1.100n \ln n$	$1.333n$	$1.600n \ln n$
3	$1.500n$	$1.385n \ln n$	$1.200n$	$1.108n \ln n$	$1.700n$	$1.569n \ln n$
4	$1.800n$	$1.403n \ln n$	$1.567n$	$1.221n \ln n$	$2.133n$	$1.662n \ln n$
5	$2.000n$	$1.379n \ln n$	$1.810n$	$1.248n \ln n$	$2.405n$	$1.658n \ln n$
6	$2.286n$	$1.435n \ln n$	$2.125n$	$1.334n \ln n$	$2.750n$	$1.726n \ln n$
7	$2.500n$	$1.455n \ln n$	$2.361n$	$1.374n \ln n$	$3.000n$	$1.746n \ln n$
8	$2.778n$	$1.519n \ln n$	$2.656n$	$1.452n \ln n$	$3.311n$	$1.810n \ln n$
9	$3.000n$	$1.555n \ln n$	$2.891n$	$1.499n \ln n$	$3.555n$	$1.843n \ln n$
15	$4.500n$	$1.890n \ln n$	$4.434n$	$1.862n \ln n$	$5.132n$	$2.156n \ln n$
31	$8.500n$	$2.779n \ln n$	$8.468n$	$2.769n \ln n$	$9.193n$	$3.006n \ln n$

benefits from using more than one pivot only with respect to scanned elements, but not with respect to the average number of write accesses and assignments.

Running Time Implications. We now ask what the considerations made so far mean for empirical running time. Since each memory access, even if it can be served from L1 cache, is much more expensive than other operations like simple subtraction, addition, or assignments on or between registers, the results for the cost measures scanned elements show that there are big differences in the time the CPU has to wait for memory between multi-pivot quicksort algorithms.[6] If in addition writing an element back into cache/memory is more expensive than reading from the cache (as it could happen with the "write-through" cache strategy), then the calculations show that we should not expect advantages of multi-pivot quicksort algorithms over classical quicksort in terms of memory behavior. However, cache architectures in modern CPUs apply the "write-back" strategy which does not add a penalty to running time for writing into memory.

As is well known from classical quicksort and dual-pivot quicksort, the influence of lower order terms cannot be neglected for real-world values of n. Next, we will validate our findings for practical values of n.

Empirical Validation. We implemented Algorithm 1 and ran it for different input lengths and pivot numbers. In the experiments, we sorted inputs of size 2^i with $9 \leq i \leq 27$. Each data point is the average over 600 trials. For measuring cache misses we used the "performance application programming interface" (PAPI), which is available at `http://icl.cs.utk.edu/papi/`.

Intuitively, fewer scanned elements should yield better cache behavior when memory accesses are done "scan-like" as in the algorithms considered here. The argument used in [LaMarca and Ladner 1999] and [Kushagra et al. 2014] is as follows: When each of the m cache memory blocks holds exactly B keys, then a scan of n' array cells (that have never been accessed before) incurs $\lceil n'/B \rceil$ cache misses. Now we check

[6] As an example, the Intel i7 used in our experiments needs at least 4 clock cycles to read from memory if the data is in L1 cache and its physical address is known. If the data is in L2 cache but not in L1 cache, there is an additional penalty of 6 clock cycles. On the other hand, three (data-independent) `MOV` operations between registers on the same core can be made in 1 clock cycle. See [Fog 2014] for more details.

tab:cache:misses:partitioning

Table III. Cache misses incurred by Algorithm 1 ("Exchange$_k$") in a single partitioning step. All values are averaged over 600 trials.

Algorithm	Exchange$_1$	Exchange$_2$	Exchange$_5$	Exchange$_9$
avg. L1 misses / n	0.125	0.163	0.25	0.378

tab:cache:misses

Table IV. Average number of L1/L2 cache misses compared to the average number of scanned elements for sorting inputs of size $n = 2^{27}$. Cache misses are scaled by $n \ln n$ and are averaged over 600 trials. In parentheses, we show the ratio to the best algorithmic variant of Algorithm 1 w. r. t. memory/cache behavior ($k = 5$), calculated from the non-truncated experimental data.

Algorithm	$\mathrm{E}(\mathrm{C}_n^{\mathrm{se}})$	**L1 Cache Misses**	**L2 Cache Misses**
Exchange$_1$	$2.000n \ln n$ (+ 45.0%)	$0.140n \ln n$ (+ 48.9%)	$0.0241n \ln n$ (+263.1%)
Exchange$_2$	$1.600n \ln n$ (+ 16.0%)	$0.110n \ln n$ (+ 16.9%)	$0.0124n \ln n$ (+ 86.8%)
Exchange$_3$	$1.385n \ln n$ (+ 0.4%)	$0.096n \ln n$ (+ 1.3%)	$0.0080n \ln n$ (+ 19.8%)
Exchange$_5$	$1.379n \ln n$ (—)	$0.095n \ln n$ (—)	$0.0067n \ln n$ (—)
Exchange$_7$	$1.455n \ln n$ (+ 5.5%)	$0.100n \ln n$ (+ 5.3%)	$0.0067n \ln n$ (+ 0.7%)
Exchange$_9$	$1.555n \ln n$ (+ 12.8%)	$0.106n \ln n$ (+ 12.2%)	$0.0075n \ln n$ (+ 12.9%)

whether the assertion that partitioning an input of n elements using Algorithm 1 incurs $\lceil \mathrm{E}(\mathrm{P}_n^{\mathrm{se}})/B \rceil$ cache misses is justifiable. (Recall that $\mathrm{E}(\mathrm{P}_n^{\mathrm{se}})$ is the average number of scanned elements during partitioning.) In the experiment, we partitioned 600 inputs consisting of $n = 2^{27}$ items using Algorithm 1, for $1, 2, 5,$ and 9 pivots. The measurements with respect to L1 cache misses are shown in Table III. In our setup, each L1 cache line contains 8 elements. So, Algorithm 1 should theoretically incur $0.125n$, $0.166n$, $0.25n$, and $0.375n$ L1 cache misses for $k \in \{1, 2, 5, 9\}$, respectively. The results from Table III show that the empirical measurements are very close to these values.

Table IV shows the exact measurements regarding L1 and L2 cache misses for sorting 600 random inputs consisting of $n = 2^{27}$ elements using Algorithm 1 and relates them to each other. The figures indicate that the relation with respect to the measured number of L1 cache misses of the different algorithms reflect their relation with respect to the average number of scanned elements *very well*. However, while the average number of cache misses correctly reflects the relative relations, the measured values (scaled by $n \ln n$) are lower than we would expect by simply dividing $\mathrm{E}(\mathrm{C}_n^{\mathrm{se}})$ by the block size B. We suspect this is due to (i) the influence of lower order terms and (ii) array segments considered in the recursion already being present in cache. In summary, scanned elements are a suitable cost measure to predict the L1 cache behavior of Algorithm 1. However, this is not true with regard to L2 cache behavior of these algorithms, as shown in Table IV.

Figure 8 shows the measurements we got with regard to the average number of assignments. We see that the measurements agree with our theoretical study (*cf.* Table II). In particular, lower order terms seem to have low influence on the sorting cost.

7.7. Comparison with Other Partitioning Algorithms

Here we compare Algorithm 1 to two algorithms known from the literature that can be used as partitioning algorithms and exhibit good runtime behavior under certain circumstances. The big difference to Algorithm 1 is that these algorithms work in a

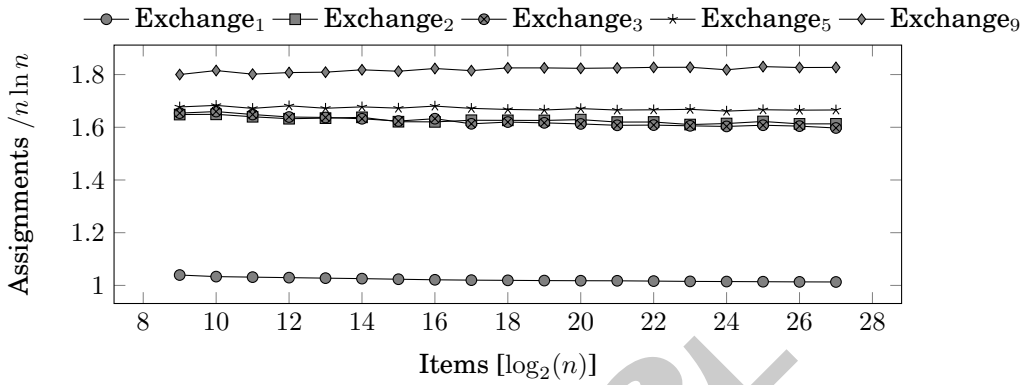

Fig. 8. The average number of assignments for sorting a random input consisting of n elements using Algorithm 1 ("Exchange$_k$") for certain values of k. Each data point is the average over 600 trials.

fig:assignments

two-pass fashion; in the first pass, they classify the input elements (to find out group sizes), in the second pass, they produce the actual partition.

We study the following adaption of Algorithm 4.1 from [McIlroy et al. 1993]. For each $k \geq 1$ we consider an algorithm *Permute*$_k$. This algorithm carries out an in-place permutation, and it works in the following way. Suppose the group sizes are $a_0, \ldots, a_k$. For each $h \in \{0, \ldots, k\}$ let $s_h = k + 1 + \sum_{0 \leq i \leq h-1} a_i$. Let $s_{k+1} = n + 1$. Then the segment in the array which (at the end) will contain the elements of group A_h in the partition is $A[s_h..s_{h+1} - 1]$. For each group A_h, $h \in \{0, \ldots, k\}$, the algorithm uses two variables. The variable c_h ("count") contains the number of elements in group A_h that have not been seen so far. (Of course, initially $\mathsf{c}_h = a_h$.) The variable o_h ("offset") contains the largest index where the algorithm has made sure that $A[s_h..\mathsf{o}_h - 1]$ only contains A_h-elements. Initially, $\mathsf{o}_h = s_h$. The algorithm uses one index j where initially $j = k + 1$. In one round, the algorithm scans the array from left to right until it finds a misplaced element at $A[j]$ with $s_h \leq j \leq s_{h+1} - 1$. Let this element be x and suppose x belongs to group $\mathrm{A}_{h'}, h' \in \{0, \ldots, k\}$. The algorithm now shifts elements in a cyclic fashion (without using extra space) to move x to a final location and write an A_h-element into $A[j]$. Technically, the algorithm repeats the following until it writes an element into $A[j]$: Scan the array from $A[\mathsf{o}_{h'}]$ to the right until a misplaced element y is reached, say, at $A[j']$. (Note that $j' \leq s_{h'+1} - 1$.) Assume y belongs to group $\mathrm{A}_{h''}$. Write x into $A[j']$. If $h'' \neq h$, set $h' := h''$ and $x := y$ and continue the loop. Otherwise write y into $A[j]$, which ends the round. Now some elements have been moved to final locations, some offsets have changed, and a new round starts. These rounds are iterated until no misplaced elements are left. Pseudocode for the algorithm is shown as Algorithm 2. An example for its memory layout is given in Figure 9 and an example for one round in the algorithm is shown in Figure 10.

We also consider a variant of Algorithm 2 we call "Copy$_k$". This algorithm was the basic partitioning algorithm in the "super scalar sample sort algorithm" of Sanders and Winkel [2004]. It uses the same offset values as Algorithm 2. Instead of carrying out an in-place permutation it allocates a new array and produces the partition by sweeping over the input array, copying elements to a final position in the new array using these offsets. So, this algorithm needs at least twice as much space as the input length. Pseudocode for this algorithm is given as Algorithm 3.

We now give a short analysis of the average number of scanned elements, write accesses, and assignments in Algorithm 2 and Algorithm 3, respectively.

Algorithm 2 Permute elements in-place to produce a partition

procedure *Permute*$_k$($A[1..n]$)
Requires: Segment sizes are $a_0, \ldots, a_k$.

```
1:  for h from 0 to k do
2:      c_h ← a_h; o_h ← k + 1 + Σ_{i=0}^{h−1} a_i;
3:  for h from 0 to k − 1 do
4:      while c_h > 0 do
5:          while A[o_h] belongs to group A_h do                ▷ Find misplaced element
6:              o_h++; c_h−−;
7:          if c_h = 0 then
8:              break;
9:          home ← o_h;
10:         prev ← home;
11:         x ← A[prev];
12:         while true do                                       ▷ Move elements cyclicly
13:             A_g ← Group of x;
14:             while A[o_g] belongs to group A_g do            ▷ Skip non-misplaced elements
15:                 o_g++; c_g−−;
16:             next ← o_g; o_g++; c_g−−;
17:             prev ← next;
18:             if home ≠ prev then
19:                 r ← A[next]; A[next] ← x; x ← r;
20:             else
21:                 A[prev] ← x;
22:                 break;
```

algo:permute:k

Algorithm 3 Copy elements to produce a partition

procedure *Copy*$_k$($A[1..n]$)
Requires: Segment sizes are $a_0, \ldots, a_k$.

```
1:  for h from 0 to k do
2:      o_h ← k + 1 + Σ_{i=0}^{h−1} a_i;
3:  allocate a new array B[k + 1..n];
4:  for i from k + 1 to n do
5:      A_p ← group of A[i];
6:      B[o[p]] ← A[i];
7:      o[p]++;
8:  Copy the content of B to A;
```

algo:copy:k

First we consider Algorithm 2. To find out the group sizes, an actual partitioning algorithm has to scan the whole array. We count one element scan, no write access and no assignment for each element. During rearranging the input, each element that is already at a final location is scanned once and involves no write access or assignment (Line 4 and Line 13). Each element that has to be moved involves one scan, one write access (to its final location) and two assignments (one for storing its value into variable x, one for writing it into its final location). Note that we do not count the assignment "$x \leftarrow r$", since it involves two variables that should be in registers and no memory access is involved. Moreover, to connect element scans to cache misses, we assume here that an array cell is never evicted from cache between the moment it was accessed for the

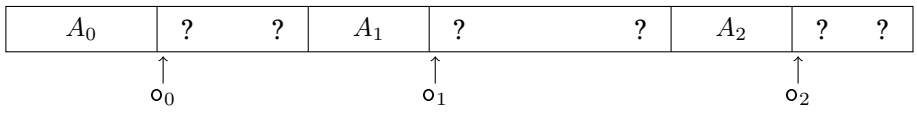

Fig. 9. General memory layout of Algorithm 2 for $k = 2$. fig:permute:k

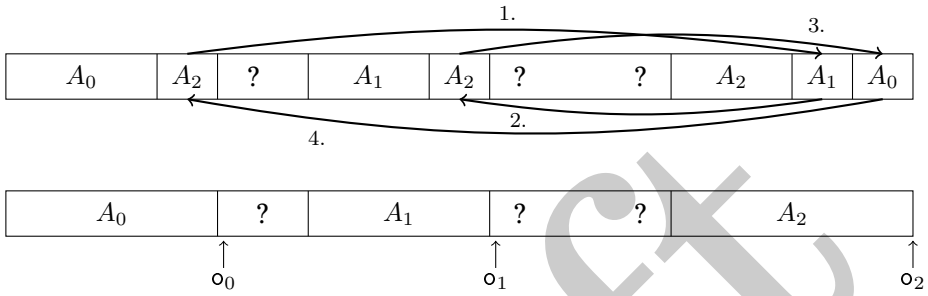

Fig. 10. Top: Example for the cyclic rotations occurring in one round of Algorithm 2 starting from the example given in Figure 9. First, the algorithm finds an A_2-element, which is then moved into the A_2-segment (1.), replacing an A_1-element which is moved into the A_1-segment (2.). It replaces an A_2-element that is moved to replace the next misplaced element in the A_2-segment, an A_0 element (3.). This element is then moved to the A_0-segment (4.), overwriting the misplaced A_2-element, which ends the round. Bottom: Memory layout and offset indices after moving the elements from the example. fig:permute:k:one:ro

first time and the moment when a final element is written into it. We will discuss this assumption later.

By a simple calculation it follows that on average there are $k(n-k)/(k+2)$ elements that have to be moved. Hence,

$$\begin{aligned}\mathbf{E}\big(\mathrm{P}_n^{\mathrm{se}}\big) &= 2(n-k),\\ \mathbf{E}\big(\mathrm{P}_n^{\mathrm{wa}}\big) &= \frac{k}{k+2}\cdot(n-k),\\ \mathbf{E}\big(\mathrm{P}_n^{\mathrm{as}}\big) &= \frac{2k}{k+2}\cdot(n-k).\end{aligned} \tag{22}$$

eq:assignments:permu

The analysis of Algorithm 3 is even simpler. Again, we count one element scan, no write access and no assignment for each element for finding out group sizes. It makes exactly $2(n-k)$ element scans and $n-k$ write accesses and assignments to rearrange the input (Line 5 in Algorithm 3). In addition, we charge $2(n-k)$ element scans and $n-k$ write accesses and assignments for copying the input back (Line 7 in Algorithm 3). So, we get

$$\begin{aligned}\mathbf{E}\big(\mathrm{P}_n^{\mathrm{se}}\big) &= 5(n-k),\\ \mathbf{E}\big(\mathrm{P}_n^{\mathrm{wa}}\big) &= \mathbf{E}\big(\mathrm{P}_n^{\mathrm{as}}\big) = 2(n-k).\end{aligned} \tag{23}$$

eq:assignments:copy

Table V shows the total sorting cost of these two algorithms for certain pivot numbers using the formulae from above and (3). Comparing with Table II, we observe the following. In general, Algorithm 2 has lower cost than Algorithm 3. Both algorithms are (asymptotically) worse than Algorithm 1 for very small values of k. When k becomes larger, the total sorting cost decreases (asymptotically). Already for 7 pivots, Algorithm 2 has (asymptotically) lower cost than the best possible pivot choices in Algorithm 1. The same is true for Algorithm 3, but it has to use more, e. g., 127 pivots to achieve this improvement over Algorithm 1. So, both algorithms have lower partitioning cost than Algorithm 1. We will see in Section 9 how these algorithms compete with Algorithm 1 with regard to running time. In short, these algorithms allow for faster sorting times,

tab:opt:cost:permute:copy

Table V. Average total sorting cost w. r. t. scanned elements, write accesses, and assignments of Algorithm 2 and Algorithm 3 for $k \in \{1, 3, 7, 15, 31, 127\}$.

k	$E(C_n^{se})$ (Algorithm 2)	$E(C_n^{wa})$ (Algorithm 2)	$E(C_n^{as})$ (Algorithm 2)	$E(C_n^{se})$ (Algorithm 3)	$E(C_n^{wa})$ (Algorithm 3)	$E(C_n^{as})$ (Algorithm 3)
1	$4.000n \ln n$	$0.667n \ln n$	$1.333n \ln n$	$10.00n \ln n$	$4.000n \ln n$	$4.000n \ln n$
3	$1.846n \ln n$	$0.554n \ln n$	$1.108n \ln n$	$4.615n \ln n$	$1.846n \ln n$	$1.846n \ln n$
7	$1.164n \ln n$	$0.453n \ln n$	$0.906n \ln n$	$2.911n \ln n$	$1.164n \ln n$	$1.164n \ln n$
15	$0.840n \ln n$	$0.371n \ln n$	$0.742n \ln n$	$2.100n \ln n$	$0.840n \ln n$	$0.840n \ln n$
31	$0.654n \ln n$	$0.307n \ln n$	$0.614n \ln n$	$1.635n \ln n$	$0.654n \ln n$	$0.654n \ln n$
127	$0.451n \ln n$	$0.222n \ln n$	$0.444n \ln n$	$1.128n \ln n$	$0.451n \ln n$	$0.451n \ln n$

but we must store element classifications from the initial classification step, which requires using additional memory in both algorithms.

Or classify twice, which is too slow.

The analysis of Algorithm 2 and Algorithm 3 showed that the respective sorting cost decreases with an increasing number of pivots. Of course, there are drawbacks of using many pivots, which we discuss now. From a theoretical point of view, we were only interested in the leading term of the average sorting cost. The cost of sorting a sample to pick the pivots should have a big influence on the lower order terms and thus influence the cost for real-world input lengths for a large number of pivots. From an architectural point of view, using many pivots means that many different areas of the array are accessed in succession (Line 14 in Algorithm 2 and Line 6 in Algorithm 3). Considering for example the Intel i7 processor used for the experiments, the L1 cache in it can store 32kB of data where each cache line can hold eight 64bit integers. This means that there are 512 different cache lines and thus at most 512 different memory segments can be accommodated at once. Since both Algorithm 2 and Algorithm 3 access memory segments depending on the outcome of classifications, their access patterns are almost fully random. (We know from Section 3 that dependencies among classifications introduce only lower order error terms.) So, for both algorithms the size of the L1 cache restricts the maximum number of pivots. In experiments we found that for up to 127 pivots there are no surprising effects and accesses that are suppossed to be served from L1 cache were served accordingly. Another cache structure important for the discussion is the *translation lookaside buffer* (TLB) that translates between virtual memory addresses (used in a process) and physical memory addresses. Each memory address used in a process must be translated to its physical address; the TLB is the cache structure that speeds up this translation. (If a "TLB miss" happens, a tree walk must be started to find out the physical memory address.) This address translation is even more important when processes run inside virtual machines because then the translation has to happen twice. Only recently, a foundation for the theoretical study of these effects has been developed by Jurkiewicz and Mehlhorn [2014]. As an example, the Intel i7 used in the experiments has a TLB that consists of two levels of 64 and 512 entries, respectively, for each core [Levinthal 2009]. In addition, there is a TLB consisting of 32 entries for large pages. However, many operation system do not support large pages. To get an impression between differences of algorithms with respect to TLB misses, we compare all algorithms to the TLB misses that happen in Algorithm 1 with one pivot for inputs of size at most 2^{27}, see Figure 11. First, for inputs of size at most 2^{23}, almost no TLB misses happen. Afterwards, all algorithms make increasingly many TLB misses. For at most 9 pivots, there is no difference between variants of Algorithm 1 to the one pivot case. Also, Algorithm 3 with up to 512 pivots makes basically the same number of TLB misses as Algorithm 1. The picture is very different for Algorithm 2. For inputs of size 2^{27} and using 127 pivots, it makes 76%

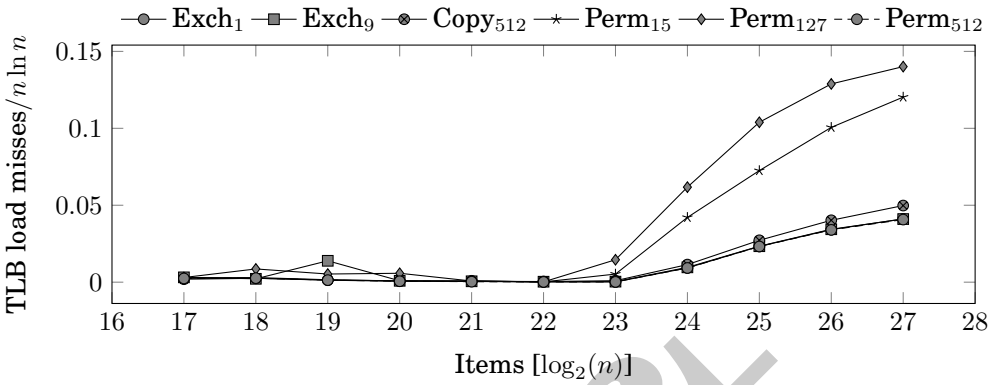

Fig. 11. TLB misses for Algorithm 1 ("Exch$_k$"), Algorithm 2 ("Perm$_k$"), and Algorithm 3 ("Copy$_k$"). Each data point is averaged over 500 trials, TLB load misses are scaled by $n \ln n$.

fig:tbl:misses:k:piv

more TLB misses than Algorithm 1, with 512 pivots it makes 244% more of them. In experiments we noticed that this has a significant negative effect to running time when comparing Algorithm 1 and Algorithm 2 as the input size increases. So, especially for Algorithm 2 the virtual address translation limits the maximum number of pivots.

7.8. Conclusion on Rearranging Elements

In this section we analyzed an algorithm (Algorithm 1) for the rearrangement problem with respect to three different cost measures. We found out that the cost measure "scanned elements" is very useful to estimate the number of cache misses. (As we will see later there is a strong correlation to running time.) With respect to the number of scanned elements, Algorithm 1 is particularly good with three or five pivots. For the cost measures "write accesses" and "assignments" we found out that the cost increases with an increasing number of pivots. We compared this algorithm with two other rearrangement algorithms from the literature. The analysis showed that they are both better than Algorithm 1 in all three cost measures, starting from 7 pivots (Algorithm 2) or 127 pivots (Algorithm 3), at the cost of higher space usage and a two-pass approach. Details of the architecture place a natural limit on the maximum number of pivots that yield efficient variants of these algorithms.

sec:pivot:sampling

8. PIVOT SAMPLING IN MULTI-PIVOT QUICKSORT

In this section we consider the benefits of sampling pivots. By "pivot sampling" we mean that we take a small, constant-sized sample of elements from the input, sort these elements, and then pick certain elements of this sorted sequence as pivots. One particularly popular strategy for classical quicksort, known as *median-of-three*, is to choose as pivot the median of a sample of three elements. From a theoretical point of view it is well known that choosing the median in a sample of $\Theta(\sqrt{n})$ elements in classical quicksort is optimal with respect to minimizing the average comparison count [Martínez and Roura 2001]. Using this sample size, quicksort achieves the (asymptotically) best possible average comparison count of $1.4426..n \ln n + O(n)$ comparisons on average. For dual-pivot quicksort, Wild, Nebel, and Martínez [2015] showed in recent work that no matter how well the pivots are chosen, Yaroslavskiy's algorithm makes at least $1.49..n \ln n + O(n)$ comparisons on average. Aumüller and Dietzfelbinger [2015] demonstrated that this is not an inherent limitation of the dual-pivot quicksort approach. Using the simple strategy of *always comparing with the largest pivot first*, they proved that choosing

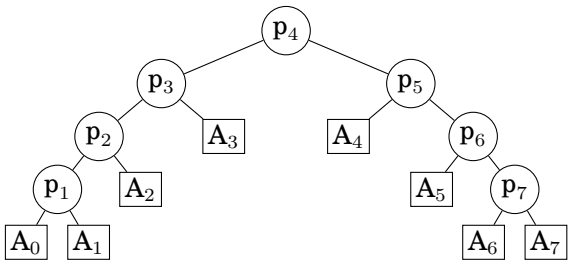

Fig. 12. The extremal tree for seven pivots.

fig:extremal:comp:tre

as pivots the elements of rank $n/4$ and $n/2$ makes it again possible to achieve the minimum possible comparison count for comparison-based sorting algorithms.

Here we study two different sampling scenarios for multi-pivot quicksort. First we develop formulae to calculate the average number of comparisons and the average number of scanned elements for Algorithm 1 when pivots are chosen from a small sample. Example calculations demonstrate that the cost in both measures can be decreased by choosing pivots from a small (fixed-sized) sample. Interestingly, with respect to scanned elements the best pivot choices do not balance subproblem sizes but tend to make the middle groups, i. e., groups A_p with p close to $\lceil \frac{k+1}{2} \rceil$, larger. Then we consider a different setting in which we can choose pivots of a given rank for free. In this setting we want to find out which pivot choices minimize the respective cost. Our first result shows that if we choose an arbitrary comparison tree and use it in every classification, it is possible to choose pivots in such a way that on average we need at most $1.4426..n \ln n + O(n)$ comparisons to sort the input. The second result is that in order to minimize the average number of scanned elements, one particular pivot choice provides minimal sorting cost. In contrast to the results of the previous section we show that with these pivot choices, the average number of scanned elements decreases with a growing number of pivots. (Recall that when choosing pivots directly from the input, the five-pivot quicksort algorithm based on Algorithm 1 has minimal cost.) From these calculations we also learn which comparison tree (among the exponentially many available) has lowest cost when considering as cost measure the sum of the number of comparisons and the number of scanned elements. In contrast to intuition, the balanced comparison tree, in which all leaves are as even in depth as possible, has non-optimal cost. The best choice under this cost measure is to use the comparison tree which uses as root pivot p_m with $m = \lceil \frac{k+1}{2} \rceil$. In its left subtree, the node labeled with pivot p_i is the left child of the node labeled with pivot p_{i+1} for $1 \le i \le m-1$. (So, the inner nodes in its left subtree are a path $(\mathrm{p}_{m-1}, \ldots, \mathrm{p}_1)$.) Analogously, in its right subtree, the node labeled with pivot p_{i+1} is the right child of the node labeled with pivot p_i for $m \le i \le k-1$. For given $k \ge 1$, we call this tree the *extremal tree for k pivots*. In Figure 12 we see an example for the extremal tree for seven pivots.

General Structure of a Multi-Pivot Quicksort Algorithm Using Sampling. We generalize a multi-pivot quicksort algorithm in the following way. (This description is analogous to those in [Hennequin 1991] for multi-pivot quicksort and [Nebel et al. 2015] for dual-pivot quicksort.) For a given number $k \ge 1$ of pivots, we fix a vector $\mathbf{t} = (t_0, \ldots, t_k) \in \mathbb{N}^{k+1}$. Let $\kappa := \kappa(\mathbf{t}) = k + \sum_{0 \le i \le k} t_i$ be the number of samples.[7])

[7]The notation differs from [Hennequin 1991] and [Nebel et al. 2015] in the following way: This paper focuses on a "pivot number"-centric approach, where the main parameter is k, the number of pivots. The other papers focus on the parameter s, the number of element groups. In particular, it holds $s = k + 1$. The sample size κ is denoted k in these papers.

Assume that an input of n elements residing in an array $A[1..n]$ is to be sorted. If $n \leq \kappa$, sort A directly. Otherwise, sort the first κ elements and then set $p_i = A[i + \sum_{j<i} t_j]$, for $1 \leq i \leq k$. Next, partition the input $A[\kappa + 1..n]$ with respect to the pivots $p_1, \ldots, p_k$. Subsequently, move the elements residing in $A[1..\kappa]$ to correct final locations. This is possible by using a constant number of rotations. Finally, sort the $k + 1$ subproblems recursively.

The sampling technique described above does not preserve randomness in subproblems, because some elements have already been sorted during the pivot sampling step. For the analysis, we ignore that the unused samples have been seen and get only an estimate on the sorting cost. A detailed analysis of this situation for dual-pivot quicksort is given in [Nebel et al. 2015], and the same methods that were proposed in their paper can be used in the multi-pivot quicksort case, too.

The Generalized Multi-Pivot Quicksort Recurrence. For a given sequence $\mathbf{t} = (t_0, \ldots, t_k) \in \mathbb{N}^{k+1}$, we define $H(\mathbf{t})$ by

$$H(\mathbf{t}) = \sum_{i=0}^{k} \frac{t_i + 1}{\kappa + 1}(H_{\kappa+1} - H_{t_i+1}). \tag{24}$$ eq:entropy

Let P_n denote the random variable which counts the cost of a single partitioning step, and let C_n denote the cost over the whole sorting procedure. In general, we get the recurrence

$$\mathrm{E}(\mathrm{C}_n) = \mathrm{E}(\mathrm{P}_n) + \sum_{a_0+\cdots+a_k=n-k} (\mathrm{E}(\mathrm{C}_{a_0}) + \cdots + \mathrm{E}(\mathrm{C}_{a_k})) \cdot \Pr(\langle a_0, \ldots, a_k \rangle), \tag{25}$$ eq:sampling:recurren

where $\langle a_0, \ldots, a_k \rangle$ is the event that the group sizes are exactly $a_0, \ldots, a_k$. The probability of this event for a given vector $\mathbf{t}$ is

$$\frac{\binom{a_0}{t_0} \cdots \binom{a_k}{t_k}}{\binom{n}{\kappa}}.$$

For the following discussion, we re-use the result of Hennequin [1991, Proposition III.9] which says that for fixed k and $\mathbf{t}$ and average partitioning cost $a \cdot n + O(1)$ recurrence (25) has the solution

$$\frac{a}{H(\mathbf{t})} n \ln n + O(n). \tag{26}$$ eq:quicksort:recurre

The Average Comparison Count Using a Fixed Comparison Tree. Fix a vector $\mathbf{t} \in \mathbb{N}^{k+1}$. First, observe that for each $i \in \{0, \ldots, k\}$ the expected number of elements belonging to group A_i is $\frac{t_i+1}{\kappa+1}(n - \kappa)$. If the $n - \kappa$ remaining input elements are classified using a fixed comparison tree λ, the average comparison count for partitioning (cf. (5)) is

$$(n - \kappa) \cdot \sum_{i=0}^{k} \mathrm{depth}_\lambda(\mathrm{A}_i) \cdot \frac{t_i + 1}{\kappa + 1}. \tag{27}$$ eq:comp:count:sampli

The Average Number of Scanned Elements of Algorithm 1. Fix a vector $\mathbf{t} \in \mathbb{N}^{k+1}$ and let $m = \lceil \frac{k+1}{2} \rceil$. Arguments analogous to the ones presented in the previous section, see (15), show that the average number of scanned elements of Algorithm 1 is

$$\begin{cases} n \cdot \sum_{i=1}^{m} i \cdot \left(\frac{t_{m-i}+t_{k-m+i}+2}{\kappa+1} \right) + O(1), & \text{for } k \text{ odd}, \\ n \cdot (m+1) \cdot \frac{t_0+1}{\kappa+1} + n \cdot \sum_{i=1}^{m} i \cdot \left(\frac{t_{m+1-i}+t_{k-m+i}+2}{\kappa+1} \right) + O(1), & \text{for } k \text{ even}. \end{cases} \tag{28}$$ eq:access:count:samp

Next, we will use these formulae to give some example calculations for small sample sizes.

Optimal Pivot Choices for Small Sample Sizes. Table VI contains the lowest possible cost for a given number of pivots and a given number of sample elements. We consider three different cost measures: the average number of comparisons, the average number of scanned elements, and the sum of these two costs. Additionally, it contains the t-vector and the comparison tree that achieves this value. (Of course, each comparison tree yields the same number of scanned elements. Consequently, no comparison tree is given for the best t-vector w. r. t. scanned elements.)

Looking at Table VI, we make the following observations. Increasing the sample size for a fixed number of pivots decreases the average cost significantly, at least asymptotically. Interestingly, to minimize the average number of comparisons, the best comparison tree is not always the one that minimizes the depth of the tree, e. g., see the best comparison tree for 5 pivots and 6 additional samples. However, for most situation it has minimal cost. To minimize the number of scanned elements, the groups in the middle should be made larger. The extremal tree provides the best possible *total cost*, summing up comparisons and scanned elements. Compared to the sampling choices that minimize the number of scanned elements, the sampled elements are slightly less concentrated around the middle element groups. This provides first evidence that the extremal tree is the best possible comparison tree for a given number of pivots with respect to the total cost.

With regard to the question of the best choice of k, Table VI shows that it is not possible to give a definite answer. All of the considered pivot numbers and additional sampling elements make it possible to decrease the total average sorting cost to around $2.6n \ln n$ using only a small sample.

Next, we will study the behavior of these cost measures when choosing the pivots is for free.

Optimal Pivot Choices. We now consider the following setting. We assume that for a random input of n elements[8] we can choose (for free) k pivots w. r. t. a vector $\tau = (\tau_0, \dots, \tau_k)$ such that the input contains exactly $\tau_i n$ elements from group A_i, for $i \in \{0, \dots, k\}$. By definition, we have $\sum_{0 \le i \le k} \tau_i = 1$. This setting was studied in [Martínez and Roura 2001; Nebel et al. 2015] as well.

We make the following preliminary observations. In our setting, the average number of comparisons *per element* (see (27)) becomes

$$c_\tau := \sum_{i=0}^{k} \mathrm{depth}_t(\mathrm{A}_i) \cdot \tau_i, \tag{29}$$

eq:sampling:comparisc

and the average number of scanned elements per element (see (28)) is

$$a_\tau := \begin{cases} \sum_{i=1}^{m-1} i \cdot (\tau_{m-i-1} + \tau_{k-m+i+1}), & \text{for } k \text{ odd}, \\ m \cdot \tau_0 + \sum_{i=1}^{m-1} i \cdot (\tau_{m-i} + \tau_{k-m+i+1}), & \text{for } k \text{ even}. \end{cases} \tag{30}$$

eq:sampling:mem:acces

Furthermore, using that the κth harmonic number H_κ is approximately $\ln \kappa$, we get that $H(\tau)$ from (24) converges to the entropy of τ

$$\mathcal{H}(\tau) := -\sum_{i=0}^{k} \tau_i \ln \tau_i.$$

[8] We disregard the k pivots in the following discussion.

Table VI. Best sampling and comparison tree choices for a given number k of pivots and a given sample size (in addition to pivots). A comparison tree is presented as the output of the preorder traversal of its inner nodes. Since the number of scanned elements is independent of the used comparison tree, no comparison tree is given for this cost measure.

tab:sampling:cost

k	Add. Samples	Cost measure	**Best candidate** (cost, t, tree)
3	0	comparisons	$1.846n \ln n$, $(0,0,0,0)$, $[2,1,3]$
		scanned elements	$1.385n \ln n$, $(0,0,0,0)$, —
		cmp + scanned elements	$3.231n \ln n$, $(0,0,0,0)$, $[2,1,3]$
	4	comparisons	$1.642n \ln n$, $(1,1,1,1)$, $[2,1,3]$
		scanned elements	$1.144n \ln n$, $(0,2,2,0)$, —
		cmp + scanned elements	$2.874n \ln n$, $(1,1,1,1)$, $[2,1,3]$
	8	comparisons	$1.575n \ln n$, $(2,2,2,2)$, $[2,1,3]$
		scanned elements	$1.098n \ln n$, $(1,3,3,1)$, —
		cmp + scanned elements	$2.745n \ln n$, $(1,3,3,1)$, $[2,1,3]$
	16	comparisons	$1.522n \ln n$, $(4,4,4,4)$, $[2,1,3]$
		scanned elements	$1.055n \ln n$, $(2,6,6,2)$, —
		cmp + scanned elements	$2.627n \ln n$, $(3,5,5,3)$, $[2,1,3]$
5	0	comparisons	$1.839n \ln n$, $(0,0,0,0,0,0)$, $[3,2,1,4,5]$
		scanned elements	$1.379n \ln n$, $(0,0,0,0,0,0)$, —
		cmp + scanned elements	$3.218n \ln n$, $(0,0,0,0,0,0)$, $[3,2,1,4,5]$
	6	comparisons	$1.635n \ln n$, $(0,0,0,2,2,2)$, $[4,3,1,2,5]$
		scanned elements	$1.097n \ln n$, $(0,1,2,2,1,0)$, —
		cmp + scanned elements	$2.741n \ln n$, $(0,1,2,2,1,0)$, $[3,2,1,4,5]$
	12	comparisons	$1.567n \ln n$, $(1,1,4,4,1,1)$, $[3,2,1,4,5]$
		scanned elements	$1.019n \ln n$, $(0,1,5,5,1,0)$, —
		cmp + scanned elements	$2.635n \ln n$, $(1,1,4,4,1,1)$, $[3,2,1,4,5]$
7	0	comparisons	$1.746n \ln n$, $(0,0,0,0,0,0,0,0)$, $[4,2,1,3,6,5,7]$
		scanned elements	$1.455n \ln n$, $(0,0,0,0,0,0,0,0)$, —
		cmp + scanned elements	$3.201n \ln n$, $(0,0,0,0,0,0,0,0)$, $[4,2,1,3,6,5,7]$
	8	comparisons	$1.595n \ln n$, $(1,1,1,1,1,1,1,1)$, $[4,2,1,3,6,5,7]$
		scanned elements	$1.094n \ln n$, $(0,0,1,3,3,1,0,0)$, —
		cmp + scanned elements	$2.698n \ln n$, $(0,0,1,3,3,1,0,0)$, $[4,3,2,1,5,6,7]$
	16	comparisons	$1.544n \ln n$, $(2,2,2,2,2,2,2,2)$, $[4,2,1,3,6,5,7]$
		scanned elements	$1.017n \ln n$, $(0,0,2,6,6,2,0,0)$, —
		cmp + scanned elements	$2.594n \ln n$, $(0,0,2,6,6,2,0,0)$, $[4,3,2,1,5,6,7]$
9	0	comparisons	$1.763n \ln n$, $(0,0,0,0,0,0,0,0,0,0)$, $[5,3,2,1,4,7,6,8,9]$
		scanned elements	$1.555n \ln n$, $(0,0,0,0,0,0,0,0,0,0)$, —
		cmp + scanned elements	$3.318n \ln n$, $(0,0,0,0,0,0,0,0,0,0)$, $[5,3,2,1,4,7,6,8,9]$
	10	comparisons	$1.602n \ln n$, $(0,0,1,2,2,2,2,1,0,0)$, $[5,3,2,1,4,7,6,8,9]$
		scanned elements	$1.131n \ln n$, $(0,0,0,1,4,4,1,0,0,0)$, —
		cmp + scanned elements	$2.748n \ln n$, $(0,0,0,1,4,4,1,0,0,0)$, $[5,4,3,2,1,6,7,8,9]$
	20	comparisons	$1.543n \ln n$, $(1,1,2,3,3,3,3,2,1,1)$, $[5,3,2,1,4,7,6,8,9]$
		scanned elements	$1.040n \ln n$, $(0,0,0,2,8,8,2,0,0,0)$, —
		cmp + scanned elements	$2.601n \ln n$, $(0,0,1,2,7,7,2,1,0,0)$, $[5,4,3,2,1,6,7,8,9]$

In the following we want to obtain optimal choices for the vector τ that minimize the three values

$$\frac{c_\tau}{\mathcal{H}(\tau)}, \qquad \frac{a_\tau}{\mathcal{H}(\tau)}, \qquad \frac{c_\tau + a_\tau}{\mathcal{H}(\tau)}, \tag{31}$$

eq:sample:asymp

i. e., that minimize the factor in the $n \ln n$ term of the sorting cost in the respective cost measure.

We start by giving a general solution to the problem of minimizing formulae where a linear function in variables $\tau_0, \ldots, \tau_k$ over the simplex $\sum_i \tau_i = 1$ is divided by the entropy of $\tau_0, \ldots, \tau_k$, as in (31).

LEMMA 8.1. *Let $\alpha_1, \ldots, \alpha_k > 0$ be arbitrary constants. For a vector $\tau = (\tau_0, \ldots, \tau_k)$ with $\sum_i \tau_i = 1$, define the function*

$$f(\tau) = \frac{\alpha_0\tau_0 + \cdots + \alpha_k\tau_k}{\mathcal{H}(\tau)}.$$

Let x be the unique solution in $(0,1)$ of the equation

$$1 = x^{\alpha_0} + x^{\alpha_1} + \cdots + x^{\alpha_k}.$$

Then $\tau = (x^{\alpha_0}, x^{\alpha_1}, \ldots, x^{\alpha_k})$ minimizes $f(\tau)$, i.e., $f(\tau) \le f(\tau')$ over all choices τ' with $\sum_i \tau'_i = 1$. This minimum is $-1/\ln x$.

m:sol:sampling:minima

PROOF. Gibb's inequality says that for arbitrary nonnegative $\sigma_0, \ldots, \sigma_k$ with $\sum_i \sigma_i \le 1$ and arbitrary nonnegative $\tau_0, \ldots, \tau_k$ with $\sum_i \tau_i = 1$ it holds

$$\mathcal{H}(\sigma_0, \ldots, \sigma_k) \le \sum_{i=0}^{k} \sigma_i \ln\left(\frac{1}{\tau_i}\right).$$

Choose $x \in (0,1)$ such that $\sum_i x^{\alpha_i} = 1$ and set $\tau_i = x^{\alpha_i}$. According to Gibb's inequality, for arbitrary $\sigma_0, \ldots, \sigma_k$ we have the bound

$$\mathcal{H}(\sigma_0, \ldots, \sigma_k) \le \sum_{i=0}^{k} \sigma_i \ln\left(\frac{1}{x^{\alpha_i}}\right) = \sum_{i=0}^{k} \sigma_i\alpha_i \ln\left(\frac{1}{x}\right).$$

So, we may conclude that for arbitrary $\sigma_0, \ldots, \sigma_k$

$$f(\sigma_0, \ldots, \sigma_k) = \frac{\sum_{i=0}^{k} \sigma_i\alpha_i}{\mathcal{H}(\sigma_0, \ldots, \sigma_k)} \ge \frac{1}{\ln\left(\frac{1}{x}\right)}.$$

Finally, observe that

$$f(\tau_0, \ldots, \tau_k) = \frac{\sum_{i=0}^{k} \tau_i\alpha_i}{\mathcal{H}(\tau_0, \ldots, \tau_k)} = \frac{\sum_{i=0}^{k} \alpha_i x^{\alpha_i}}{-\sum_{i=0}^{k} x^{\alpha_i} \ln(x^{\alpha_i})} = \frac{1}{\ln\left(\frac{1}{x}\right)}. \qquad \square$$

We first consider optimal choices for the sampling vector τ in order to minimize the average number of comparisons. The following theorem says that each comparison tree makes it possible to achieve the minimum possible sorting cost for comparison-based sorting algorithms in the considered setting.

THEOREM 8.2. *Let $k \ge 1$ be fixed. Let $\lambda \in \Lambda^k$ be an arbitrary comparison tree. Then there exists τ such that the average comparison count using comparison tree λ in each classification is $(1/\ln 2)n \ln n + O(n) = 1.4426..n \ln n + O(n)$.*

thm:opt:class

PROOF. For each $i \in \{0, \ldots, k\}$, set $\tau_i = 2^{-\mathrm{depth}_\lambda(\mathrm{A}_i)}$. First, observe that $\sum \tau_i = 1$. (This is true since every inner node in λ has exactly two children.) Starting from (27), we may calculate:

$$\sum_{i=0}^{k} \mathrm{depth}_\lambda(\mathrm{A}_i) \cdot \tau_i = \sum_{i=0}^{k} -\log(\tau_i) \cdot \tau_i = -\frac{1}{\ln 2} \sum_{i=0}^{k} \tau_i \cdot \ln(\tau_i).$$

This shows the theorem. □

We now consider the average number of scanned elements of Algorithm 1. The following theorem says that pivots should be chosen in such a way that (in the limit for $k \to \infty$) $2/3$ of the input are only scanned once (by the pointers `i` and `j`), $2/9$ should be scanned twice, and so on.

THEOREM 8.3. *Let $k \geq 1$ be fixed. Let $m = \lceil \frac{k+1}{2} \rceil$. Let τ be chosen according to the following two cases:*

(*1*) *If k is odd, let x be the unique value in $(0, 1)$ such that*

$$1 = 2(x + x^2 + \cdots + x^m).$$

Let $\tau = (x^m, x^{m-1}, \ldots, x, x, \ldots, x^m)$.

(*2*) *If k is even, let x be the unique value in $(0, 1)$ such that*

$$1 = 2(x + x^2 + \cdots + x^{m-1}) + x^m.$$

Let $\tau = (x^m, x^{m-1}, \ldots, x, x, \ldots, x^{m-1})$.

Then the average number of scanned elements using Algorithm 1 with τ is minimal over all choices of vectors τ'. For $k \to \infty$, this minimum is $1/(\ln 3) n \ln n \approx 0.91 n \ln n$ scanned elements.

thm:mem:acc

PROOF. Setting the values $\alpha_0, \ldots, \alpha_k$ in Lemma 8.1 according to Equation (30) shows that the choices for τ are optimal with respect to minimizing the average number of scanned elements. One easily checks that in the limit for $k \to \infty$ the value x in the statement is $1/3$. □

For example, in the special case of Yaroslavskiy's algorithm Nebel et al. [2015] noticed that $\tau = (q^2, q, q)$ with $q = \sqrt{2} - 1$ is the optimal pivot choice to minimize element scans. In this case, around $1.13 n \ln n$ elements are scanned on average. The minimal average number of scanned elements using Algorithm 1 for $k \in \{3, 5, 7, 9\}$ are around $0.995 n \ln n$, $0.933 n \ln n$, $0.917 n \ln n$, and $0.912 n \ln n$, respectively. Hence, already for small values of k the average number of scanned elements is close to $0.91 n \ln n$. However, from Table VI we see that for sample sizes suitable in practice, both the average comparison count and average pointer visit count are around $0.1 n \ln n$ higher than these asymptotic values.

To summarize, we learned that (i) every fixed comparison tree yields an optimal classification strategy and (ii) one specific pivot choice has the best possible average number of scanned elements in Algorithm 1. Next, we consider as cost measure the sum of the average number of comparisons and the average number of scanned elements.

THEOREM 8.4. *Let $k \geq 1$ be fixed. Let $m = \lceil \frac{k+1}{2} \rceil$. Let τ be chosen according to the following two cases:*

(*1*) *If k is odd, let x be the unique value in $(0, 1)$ such that*

$$1 = 2(x^3 + x^5 + \cdots + x^{2m-3} + x^{2m-1} + x^{2m}).$$

Let $\tau = (x^{2m}, x^{2m-1}, x^{2m-3}, \ldots, x, x, \ldots, x^{2m-3}, x^{2m-1}, x^{2m})$.

(2) *If k is even, let x be the unique value in $(0,1)$ such that*

$$1 = 2(x^3 + x^5 + \cdots + x^{2m-3}) + x^{2m-2} + x^{2m-1} + x^{2m}.$$

Let $\tau = (x^{2m}, x^{2m-1}, x^{2m-3}, \ldots, x^3, x^3, \ldots, x^{2m-3}, x^{2m-2})$.

thm:memcmp:sampling

Then the average cost using Algorithm 1 and classifiying all elements with the extremal comparison tree for k pivots using τ is minimal among all choices of vectors τ'. For $k \to \infty$ this minimum cost is about $2.38n \ln n$.

PROOF. First, observe that for the extremal comparison tree for k pivots, (29) becomes

$$c_\tau = \begin{cases} m(\tau_0 + \tau_k) + \sum_{i=1}^{m-1}(i+1)(\tau_{m-i} + \tau_{m+i-1}), & \text{for } k \text{ odd}, \\ m(\tau_0 + \tau_1) + (m-1)\tau_k + \sum_{i=1}^{m-2}(i+1)(\tau_{m-i} + \tau_{m+i-1}), & \text{for } k \text{ even}. \end{cases} \quad (32)$$

eq:proof:memcmp:1

The optimality of the τ choice in the theorem statement now follows from adding (30) to (32), and using Lemma 8.1. For $k \to \infty$, the optimal x value is the unique solution in $(0,1)$ of the equation $2x^3 + x^2 = 1$, which is about 0.6573. Thus, the total cost is about $2.38n \ln n$. □

Interestingly, the minimal total cost of $2.38n \ln n$ is only about $0.03n \ln n$ higher than adding $(1/\ln 2)n \ln n$ (the minimal average comparison count) and $0.91n \ln n$ (the minimal average number of scanned elements). Using the extremal tree is much better than, e. g., using the balanced comparison tree for $k = 2^\kappa - 1$ pivots. The minimum sorting cost using this tree is $2.489n \ln n$, which is achieved for three pivots[9]. We conjecture that the extremal tree has minimal total sorting cost, i. e., minimizes the sum of scanned elements and comparisons. We found this to be true via exhaustive search for $k \leq 9$ pivots.

Again, including scanned elements as cost measure yields unexpected results and design recommendations for engineering a sorting algorithm. Looking at comparisons, the balanced tree for $2^\kappa - 1$ pivots is the obvious comparison tree to use in a multi-pivot quicksort algorithm. Only when including scanned elements, the extremal tree shows its potential in leading to fast multi-pivot quicksort algorithms.[10]

Fußnote in Einleitung?

sec:experiments

9. EXPERIMENTS

In this section, we focus on the actual running time needed to sort a random permutation of the set $\{1, \ldots, n\}$ using multi-pivot quicksort methods. We implemented the algorithms in C++ and used *clang* in version 3.5 for compiling with the optimization flag *-O3*. Our experiments were carried out on an Intel i7-2600 at 3.4 GHz with 16 GB Ram running Ubuntu 14.10 with kernel version 3.16.0. The source code of the multi-pivot quicksort algorithms is available at http://eiche.theoinf.tu-ilmenau.de/quicksort-experiments/.

We restricted our experiments to sorting random permutations of the integers $\{1, \ldots, n\}$. We tested inputs of size 2^i, $21 \leq i \leq 27$. For each input size, we ran each algorithm on the same 600 random permutations. All figures in the following are the average over these 600 trials.

Our goal in the following is to compare empirical running times of the generic approach "k-pivot quicksort". To achieve this, all algorithms were generated automatically based on Algorithm 1, where classifications in Line 7 and Line 12 were implemented

[9]For three pivots, the extremal tree and the balanced tree are identical.

[10]Here, we want to stress that papers dealing with multi-pivot quicksort such as [Hennequin 1991; Kushagra et al. 2014; Iliopoulos 2014] do not describe the full design space for multi-pivot quicksort algorithms, but rather always assume the "most-balanced" tree is best.

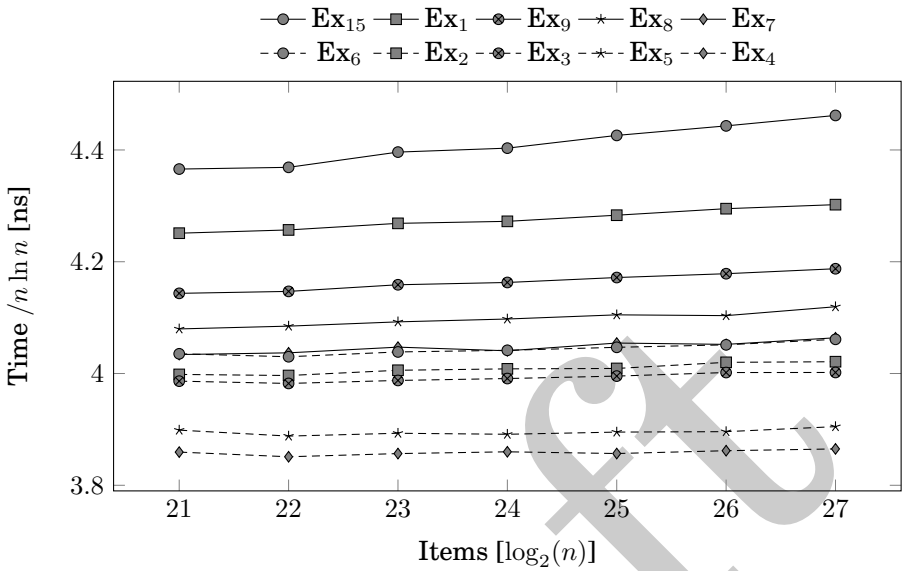

Fig. 13. Running time experiments for k-pivot quicksort algorithms based on the "Exchange$_k$" partitioning strategy. Each data point is the average over 600 trials. Times are scaled by $n \ln n$.

fig:running:times:k:

using the extremal comparison tree. The script that generates the k-pivot quicksort algorithms also takes a sampling vector τ as input to apply different pivot sampling strategies. (We remark that manually writing the algorithm code is error-prone and tedious. For example, the source code for the 9-pivot algorithm uses 376 lines of code and needs nine different rotate operations.) Subarrays of size at most 500 were sorted using the fast three-pivot quicksort algorithm of [Kushagra et al. 2014]. At the end of this section, we compare the results to Algorithm 2, Algorithm 3, and the standard introsort sorting method from C++'s standard library.

The Extremal Comparison Tree vs. the Balanced Comparison Tree. We report on the differences between the 7-pivot algorithm that uses the extremal comparison tree and the 7-pivot algorithm that uses the balanced comparison tree. For inputs of size $n = 2^{27}$, the 7-pivot quicksort algorithm using the extremal tree was at least 2% faster than the 7-pivot algorithm using the balanced tree in 95% of the runs. The difference in running time is hence statistically significant, but only minimal.

Running Times of k-pivot Quicksort Algorithms. The time measurements of our experiments can be seen in Figure 16. With respect to the average running time, we see that the variants using 4 and 5 pivots are the fastest algorithms. On average the 5-pivot algorithm is about 1% slower than the 4-pivot algorithm. However, there is no significant difference in running time between these two variants. On average, the 3-pivot and 2-pivot algorithm are 3.5% and 3.7% slower than the 4-pivot quicksort algorithm. The 6- and 7-pivot algorithms are about 5.0% slower. It follows the 8-pivot algorithm (6.5% slower), the 9-pivot algorithm (8.5% slower), classical quicksort (11.0% slower), and the 15-pivot algorithm (15.5% slower). With respect to significant differences in running time, i. e., differences observed in at least 95% of all runs, these numbers decrease by about 1–2%. We got similar results for a different setup, see Appendix B.

Comparing these running time measurements to the theoretical study we conducted in this paper, only the average number of scanned elements of an algorithm corre-

sponds (to a considerable extent) to observed running times. (The average number of comparisons decreases with the number of pivots, the average number of assignments increases with the number of pivots.) Only for scanned elements, algorithms using 3–5 pivots should be the fastest algorithms, and using more than 5 pivots should increase the running time, see Table II. We have observed this effect nicely in our running time experiments. For a large number of pivots, additional work, e. g., finding the pivots, seems to have a noticeable influence on running time. For example, according to the average number of scanned elements the 15-pivot quicksort should not be slower than classical quicksort.

Shows importance of experiments.

The Influence of Sampling to Running Time. Here we report on the influence of sampling to running time. First we want to stress that comparing the overhead incured by sorting a larger sample to the benefits of having better pivots is very difficult from a theoretical point of view because of the influence of lower-order terms to the sorting cost for real-world values of n . In our experiments we observed that samples that make the groups closer to the center larger improved the running time more significantly than balanced samples, which validates our finding from Section 8. However, the benefits of choosing pivots from a small sample to empirical running time when sorting random inputs are marginal. Compared to choosing pivots directly from the input, the largest improvement in running time were observed for the 7- and 9-pivot algorithms. For these algorithms, the running time could be improved by about 3% by choosing pivots from a small sample. Figure 14 depicts the running times we got for variants using seven pivots and different sampling vectors. In some cases, e. g., for the 4-pivot algorithm, we could not observe any improvements by sampling pivots. This does not support the hypothesis that improvements of multi-pivot quicksort algorithms are largely due to its better cache behavior, because sampling improves the cache behavior, see Table VI.[11]

Comparison with Other Methods. Finally, we report on experiments that compared the algorithms from before with other quicksort-based algorithms known from the literature. For the comparison, we used the `std::sort` implementation found in C++'s standard STL (from gcc), Yaroslavskiy's algorithm from [Nebel et al. 2015, Figure 4], the two-pivot algorithm from [Aumüller and Dietzfelbinger 2015, Algorithm 3], the three-pivot algorithm of [Kushagra et al. 2014], see [Aumüller and Dietzfelbinger 2015, Algorithm 8], and an implementation of the super scalar sample sort algorithm of Sanders and Winkel [2004] (Algorithm 3 in Section 7) with basic source code provided by Timo Bingmann. Algorithm `std::sort` is an introsort implementation, which combines quicksort with a heapsort fallback when subproblem sizes decrease too slowly. As explained in [Sanders and Winkel 2004], Algorithm 3 can be implemented to exploit *data independence* (classifications are decoupled from each other) and *predicated move instructions*, which reduce branch mispredictions in the classification step. See the source code at http://eiche.theoinf.tu-ilmenau.de/quicksort-experiments/ for details.

Figure 15 shows the measurements we got for these algorithms.[12] We see that `std::sort` is by far the slowest algorithm. Of course, it is slowed down by special precautions for inputs that are not random or have equal entries. (Such precautions have not been taken in the other implementations.) Next come the two dual-pivot quicksort algorithms. The three-pivot algorithm of [Kushagra et al. 2014] is a little bit

[11] For example, the 4-pivot algorithm without sampling incurred 10% more L1 cache misses, 10% more L2 cache misses, and needed 7% more instructions than the 4-pivot algorithm with sampling vector $(0, 0, 1, 1, 0)$. Still, it was 2% faster in experiments. The reason might be that it made 7% less branch mispredictions.

[12] We used *gcc* version 4.9 to compile Yaroslavskiy's algorithm and the algorithm of Sanders and Winkel. For both algorithms, the executable obtained by compiling with *clang* was much slower. For Yaroslavskiy's algorithm, compiling with the compiler flag *-O3* was fastest, while for the algorithm of Sanders and Winkel the flags *-O3 -funroll-loops* were fastest.

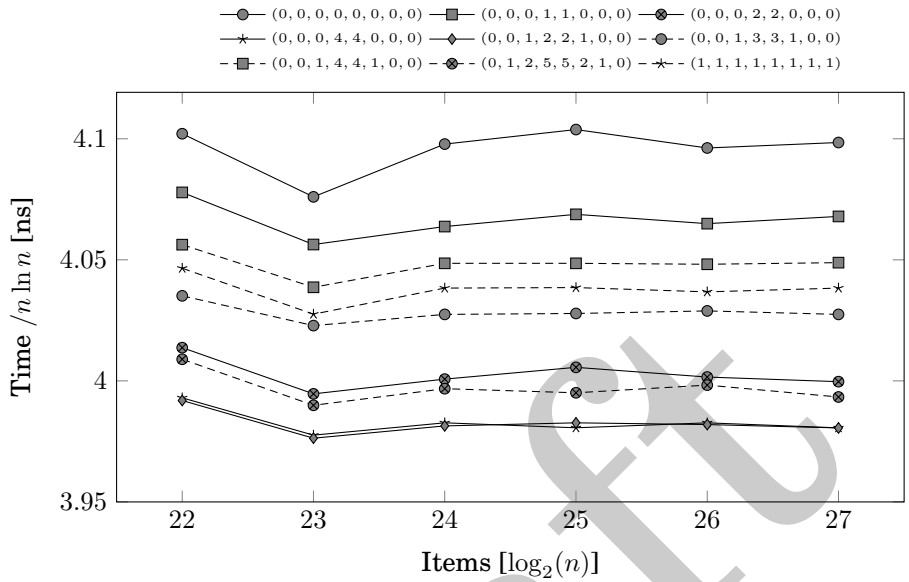

Fig. 14. Running time experiments for different sampling strategies for the 7-pivot quicksort algorithm based on the "Exchange$_7$" partitioning strategy. Each line represents the running time measurements obtained with the given τ sampling vector. Each data point is the average over 600 trials. Times are scaled by $n \ln n$.

fig:running:times:7:

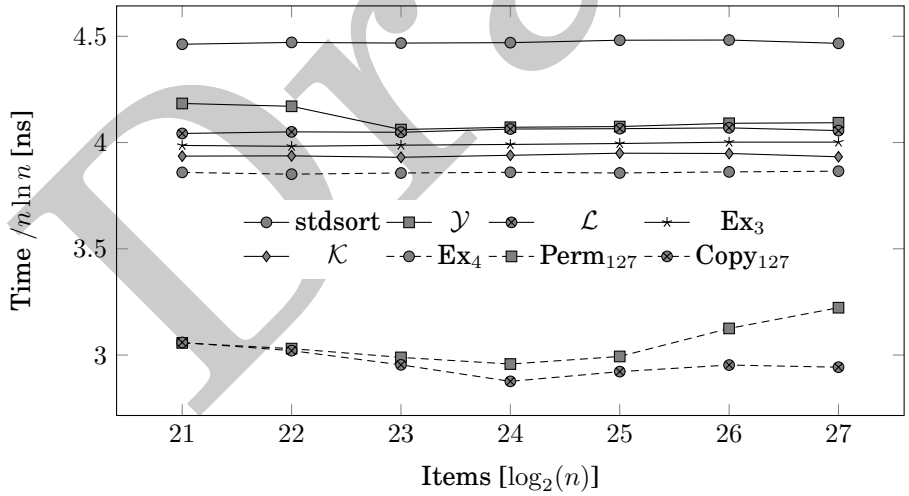

Fig. 15. Running time experiments for different quicksort-based algorithms compared to the "Exchange$_k$" partitioning strategies Ex$_3$ and Ex$_4$. The plot includes the average running time of C++'s `std::sort` implementation ("stdsort"), Yaroslavskiy's algorithm ("$\mathcal{Y}$"), the dual-pivot quicksort algorithm from [Aumüller and Dietzfelbinger 2015, Algorithm 3] ("$\mathcal{L}$"), the three-pivot quicksort algorithm from Kushagra et al. ("$\mathcal{K}$"), the variant of Algorithm 2 using 127 pivots ("Permute$_{127}$") and the variant of Algorithm 3 using 127 pivots ("Copy$_{127}$"). Each data point is the average over 600 trials. Times are scaled by $n \ln n$.

fig:running:times:ot

faster than the automatically generated three-pivot quicksort algorithm, but a little slower than the automatically generated four-pivot quicksort algorithm. (This shows that the automatically generated quicksort algorithms do not suffer in performance compared to fine-tuned implementations such as Yaroslavskiy's algorithm and the

three-pivot algorithm from [Kushagra et al. 2014].) The super scalar sample sort algorithm of Sanders and Winkel is by far the fastest algorithm, being roughly 31% faster than the four-pivot algorithm. (However, it needs about twice as much space and good compiler/hardware support.) We remark that on our hardware, the variant of Algorithm 3 with 127 pivots is fastest. Using 255 pivots is slightly slower, using 511 is much slower. Moreover, Algorithm 2 using 127 pivots is slower than Algorithm 3 with 127 pivots, and makes many misses in the TLB if the input consists of more than 2^{25} items. Still, it was about 15% faster than the fastest variant of Algorithm 1, but also has to store the results of the classification step to be competitive in running time.

Conclusion on experiments?

10. CONCLUSION

In this paper we studied the design space of multi-pivot quicksort algorithms and demonstrated how to analyze their sorting cost. In the first part we showed how to calculate the average comparison count of an arbitrary multi-pivot quicksort algorithm. We described optimal multi-pivot quicksort algorithms with regard to key comparisons. It turned out that calculating their average comparison count seems difficult already for four pivots (we resorted to experiments) and that they cannot compete with simpler variants with respect to running time. Similar improvements in key comparisons can be achieved by much simpler strategies such as the median-of-k strategy for classical quicksort. In the second part we switched our viewpoint and studied the problem of rearranging entries to obtain a partition of the input. We analyzed a one-pass algorithm (Algorithm 1) with respect to the cost measures "scanned elements", "write accesses", and "assignments". For the second and third cost measure, we found that the cost increases with an increasing number of pivots. The first cost measure turned out to be more interesting. Using Algorithm 1 with three or five pivots was particularly good, and we asserted that "scanned elements" correspond to L1 cache misses in practice. Experiments revealed that there is a high correlation to observed running times. We also analyzed two algorithms (Algorithm 2 and Algorithm 3) known from the literature and found out that they have decreasing cost as the number of pivots increases, but details of the architecture place a natural limit on the maximum number of pivots. In the last part of this paper, we discussed the influence of pivot sampling to sorting cost. With respect to comparisons we showed that for every comparison tree we can describe a pivot choice such that the cost is optimal. With regard to the number of scanned elements, we noticed that pivots should be chosen such that they make groups closer to the center larger. To determine element groups, the extremal comparison tree should be used. We conjectured that this choice of tree is also best when we consider as cost the sum of comparisons and scanned elements.

For future work, it would be very interesting to see how the optimal average comparison count of k-pivot quicksort can be calculated analytically, cf. (11). With respect to the rearrangement problem, it would be interesting to identify an optimal algorithm that shifts as few elements as possible to rearrange an input. From a empirical point of view, it would be interesting to test multi-pivot quicksort algorithms at input distributions that are not random permutation, but have some kind of "presortedness" or contain equal keys.

Acknowledgements

The first two authors thank Conrado Martínez, Markus Nebel, and Sebastian Wild for interesting discussions and comments on multi-pivot quicksort. In addition, the first author thanks Timo Bingmann for providing source code and interesting discussions.

REFERENCES

AggarwalV88 Alok Aggarwal and Jeffrey Scott Vitter. 1988. The Input/Output Complexity of Sorting and Related Problems. *Commun. ACM* 31, 9 (1988), 1116–1127. DOI: http://dx.doi.org/10.1145/48529.48535

AumullerD13 Martin Aumüller and Martin Dietzfelbinger. 2013. Optimal Partitioning for Dual Pivot Quicksort. In *ICALP (1) (Lecture Notes in Computer Science)*, Fedor V. Fomin, Rusins Freivalds, Marta Z. Kwiatkowska, and David Peleg (Eds.), Vol. 7965. Springer, 33–44.

AumullerD15 Martin Aumüller and Martin Dietzfelbinger. 2015. Optimal Partitioning for Dual Pivot Quicksort. (2015). To appear in ACM Transactions on Algorithms.

BentleyM93 Jon L. Bentley and M. Douglas McIlroy. 1993. Engineering a sort function. *Software: Practice and Experience* 23, 11 (1993), 1249–1265.

BentleyS97 Jon Louis Bentley and Robert Sedgewick. 1997. Fast Algorithms for Sorting and Searching Strings. In *Proc. of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'97)*. ACM, 360–369.

Dijkstra76 Edsger W. Dijkstra. 1976. *A Discipline of Programming*. Prentice-Hall.

dp09 Devdatt P. Dubhashi and Alessandro Panconesi. 2009. *Concentration of Measure for the Analysis of Randomized Algorithms*. Cambridge University Press. I–XIV, 1–196 pages.

Fog14 Agner Fog. 2014. 4. Instruction tables. http://www.agner.org/optimize/instruction_tables.pdf. (2014).

FrigoLPR12 Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. 2012. Cache-Oblivious Algorithms. *ACM Transactions on Algorithms* 8, 1 (2012), 4. DOI: http://dx.doi.org/10.1145/2071379.2071383

GrahamKP Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. 1994. *Concrete mathematics - a foundation for computer science (2. ed.)*. Addison-Wesley.

hennequin Pascal Hennequin. 1991. *Analyse en moyenne d'algorithmes: tri rapide et arbres de recherche*. Ph.D. Dissertation. Ecole Politechnique, Palaiseau.

Hoare62 C. A. R. Hoare. 1962. Quicksort. *Comput. J.* 5, 1 (1962), 10–15.

Iliopoulos14 Vasileios Iliopoulos. 2014. A note on multipivot Quicksort. *CoRR* abs/1407.7459 (2014).

JurkiewiczM14 Tomasz Jurkiewicz and Kurt Mehlhorn. 2014. On a Model of Virtual Address Translation. *ACM Journal of Experimental Algorithmics* 19, 1 (2014). DOI: http://dx.doi.org/10.1145/2656337

KaligosiS06 Kanela Kaligosi and Peter Sanders. 2006. How Branch Mispredictions Affect Quicksort. In *Proc. of the 14th Annual European Symposium on Algorithms (ESA'06)*. Springer, 780–791. DOI: http://dx.doi.org/10.1007/11841036_69

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

Kushagra14 Shrinu Kushagra, Alejandro López-Ortiz, Aurick Qiao, and J Ian Munro. 2014. Multi-Pivot Quicksort: Theory and Experiments. In *ALENEX'14*.

LaMarcaL99 Anthony LaMarca and Richard E. Ladner. 1999. The Influence of Caches on the Performance of Sorting. *J. Algorithms* 31, 1 (1999), 66–104.

Levinthal09 David Levinthal. 2009. Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors. https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf. (2009).

MartinezNW15 Conrado Martínez, Markus E. Nebel, and Sebastian Wild. 2015. Analysis of Branch Misses in Quicksort. In *Proc. of the 12th Meeting on Analytic Algorithmics and Combinatorics (ANALCO' 15)*.

MartinezR01 Conrado Martínez and Salvador Roura. 2001. Optimal Sampling Strategies in Quicksort and Quickselect. *SIAM J. Comput.* 31, 3 (2001), 683–705.

McIlroyBM93 Peter M. McIlroy, Keith Bostic, and M. Douglas McIlroy. 1993. Engineering Radix Sort. *Computing Systems* 6, 1 (1993), 5–27.

NebelWM15 Markus E. Nebel, Sebastian Wild, and Conrado Martínez. 2015. Analysis of Pivot Sampling in Dual-Pivot Quicksort: A Holistic Analysis of Yaroslavskiy's Partitioning Scheme. *Algorithmica* (2015), 1–52. DOI: http://dx.doi.org/10.1007/s00453-015-0041-7

Rahman02 Naila Rahman. 2002. Algorithms for Hardware Caches and TLB. In *Algorithms for Memory Hierarchies, Advanced Lectures [Dagstuhl Research Seminar, March 10-14, 2002]*. 171–192. DOI: http://dx.doi.org/10.1007/3-540-36574-5_8

Roura01 Salvador Roura. 2001. Improved master theorems for divide-and-conquer recurrences. *J. ACM* 48, 2 (2001), 170–205.

SandersW04 Peter Sanders and Sebastian Winkel. 2004. Super Scalar Sample Sort. In *Proc. of the 12th Annual European Symposium on Algorithms (ESA'04)*. Springer, 784–796.

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

SedgewickEqual Robert Sedgewick. 1977. Quicksort with Equal Keys. *SIAM J. Comput.* 6, 2 (1977), 240–268.

ShepherdsonS63 John C. Shepherdson and Howard E. Sturgis. 1963. Computability of Recursive Functions. *J. ACM* 10, 2 (1963), 217–255. DOI:http://dx.doi.org/10.1145/321160.321170

Tan93 Kok-Hooi Tan. 1993. *An asymptotic analysis of the number of comparisons in multipartition quicksort*. Ph.D. Dissertation. Carnegie Mellon University.

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

nebel12 Sebastian Wild and Markus E. Nebel. 2012. Average Case Analysis of Java 7's Dual Pivot Quicksort. In *ESA'12*. 825–836.

WildNN15 Sebastian Wild, Markus E. Nebel, and Ralph Neininger. 2015. Average Case and Distributional Analysis of Dual-Pivot Quicksort. *ACM Transactions on Algorithms* 11, 3 (2015), 22.

c:recurrence:solution

A. SOLVING THE GENERAL QUICKSORT RECURRENCE

In Section 2 we have shown that the sorting cost of k-pivot quicksort follows the recurrence:

$$\mathbf{E}(C_n) = \mathbf{E}(P_n) + \frac{1}{\binom{n}{k}} \sum_{i=0}^{n-k} (k+1) \binom{n-i-1}{k-1} \mathbf{E}(C_i).$$

We will use the continuous Master theorem of Roura [2001] to solve this recurrence. For completeness, we give the CMT below:

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

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

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

$$\sum_{j=0}^{n-1} \left| w_{n,j} - \int_{j/n}^{(j+1)/n} w(z)\, dz \right| = O(n^{-d}), \tag{33}$$

eq:cmt:shape:functio

for a constant $d > 0$. Let $H := 1 - \int_0^1 z^\alpha w(z)\, dz$. Then we have the following cases:[13]

(1) If $H > 0$, then $F_n \sim t_n / H$.

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

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

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

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

thm:CMT

THEOREM A.2. *Let $\mathcal{A}$ be a k-pivot quicksort algorithm which has for each subarray of length n partitioning cost $\mathbf{E}(P_n) = a \cdot n + O(n^{1-\varepsilon})$. Then*

$$\mathbf{E}(C_n) = \frac{1}{H_{k+1} - 1} a n \ln n + O(n),$$

where $H_{k+1} = \sum_{i=1}^{k+1} (1/i)$ is the $(k+1)$st harmonic number.

[13] Let $f(n)$ and $g(n)$ be two functions. We write $f(n) \sim g(n)$ if $f(n) = g(n) + o(g(n))$. If $f(n) \sim g(n)$, we say that "*f and g are asymptotically equivalent.*"

PROOF. By linearity of expectation we may obtain a solution for the recurrence for toll function $t_{1,n} = a \cdot n$ and toll function $t_{2,n} = K \cdot n^{1-\varepsilon}$ separately and add the solutions.

For toll function $t_{1,n}$, we use the result of Hennequin [1991, Proposition III.9] that says that for partitioning cost $a \cdot n + O(1)$ we get sorting cost $\mathrm{E}(C_{1,n}) = \frac{a}{H_{k+1}-1} n \ln n + O(n)$.

For $t_{2,n}$, we apply the CMT as follows. First, observe that Recurrence (2) has weight

$$w_{n,j} = \frac{(k+1) \cdot k \cdot (n-j-1) \cdot \ldots \cdot (n-j-k+1)}{n \cdot (n-1) \cdot \ldots \cdot (n-k+1)}.$$

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

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

We note that for all $z \in [0,1]$:

$$\begin{aligned}
|n \cdot w_{n,zn} - w(z)| &\le k \cdot (k+1) \cdot \left| \frac{(n-zn-1)^{k-1}}{(n-k)^{k-1}} - (1-z)^{k-1} \right| \\
&= k \cdot (k+1) \cdot \left| \left(\frac{n(1-z)}{n-k} - \frac{1}{n-k} \right)^{k-1} - (1-z)^{k-1} \right| \\
&\le k \cdot (k+1) \cdot \left| \left(\frac{n(1-z)}{n-k} \right)^{k-1} - (1-z)^{k-1} + O\big(n^{-1}\big) \right| \\
&\le k \cdot (k+1) \cdot \left| (1-z)^{k-1} \cdot \left(\frac{1}{\left(1-\frac{k}{n}\right)^{k-1}} - 1 \right) + O\big(n^{-1}\big) \right| \\
&\le k \cdot (k+1) \cdot \left| (1-z)^{k-1} \cdot \left(\frac{1}{1-O(n^{-1})} - 1 \right) + O\big(n^{-1}\big) \right| \\
&\le k \cdot (k+1) \cdot \left| (1-z)^{k-1} \cdot O(n^{-1}) + O\big(n^{-1}\big) \right| \\
&= O(n^{-1}),
\end{aligned}$$

where we used the Binomial theorem for the asymptotic bounds.

Now we have to check (33) to see whether the shape function is suitable. We calculate:

$$\begin{aligned}
&\sum_{j=0}^{n-1}\left|w_{n,j} - \int_{j/n}^{(j+1)/n} w(z)\,\mathrm{d}z\right| \\
&= \sum_{j=0}^{n-1}\left|\int_{j/n}^{(j+1)/n} n\cdot w_{n,j} - w(z)\,\mathrm{d}z\right| \\
&\le \sum_{j=0}^{n-1}\frac{1}{n}\max_{z\in[j/n,(j+1)/n]}|n\cdot w_{n,j} - w(z)| \\
&\le \sum_{j=0}^{n-1}\frac{1}{n}\Big(\max_{z\in[j/n,(j+1)/n]}|w(j/n) - w(z)| + O\big(n^{-1}\big)\Big) \\
&\le \sum_{j=0}^{n-1}\frac{1}{n}\Big(\max_{|z-z'|\le 1/n}|w(z) - w(z')| + O\big(n^{-1}\big)\Big) \\
&\le \sum_{j=0}^{n-1}\frac{k(k+1)}{n}\Big(\max_{|z-z'|\le 1/n}\big|(1-z)^{k-1} - (1-z-1/n)^{k-1}\big| + O\big(n^{-1}\big)\Big) \\
&\le \sum_{j=0}^{n-1} O\big(n^{-2}\big) = O\big(n^{-1}\big),
\end{aligned}$$

where we again used the Binomial theorem in the last two lines.

Thus, w is a suitable shape function. Using partial integration, we see that

$$H := 1 - k(k+1)\int_0^1 z^{1-\varepsilon}(1-z)^{k-1}\,\mathrm{d}z < 0.$$

Thus, the third case of the CMT applies. Again using partial integration, we check that

$$k(k+1)\int_0^1 z(1-z)^{k-1}\,\mathrm{d}z = 1,$$

so we conclude that for toll function $t_{2,n}$ the recurrence has solution $\mathrm{E}(C_{2,n}) = O(n)$. The theorem follows by adding $\mathrm{E}(C_{1,n})$ and $\mathrm{E}(C_{2,n})$. □

app:ktinfty

B. EXPERIMENTS ON A DIFFERENT MACHINE

These experiments were carried out on an Intel Xeon E5645 at 2.4 GHz with 48 GB Ram running Ubuntu 14.04 with kernel version 3.13.0.

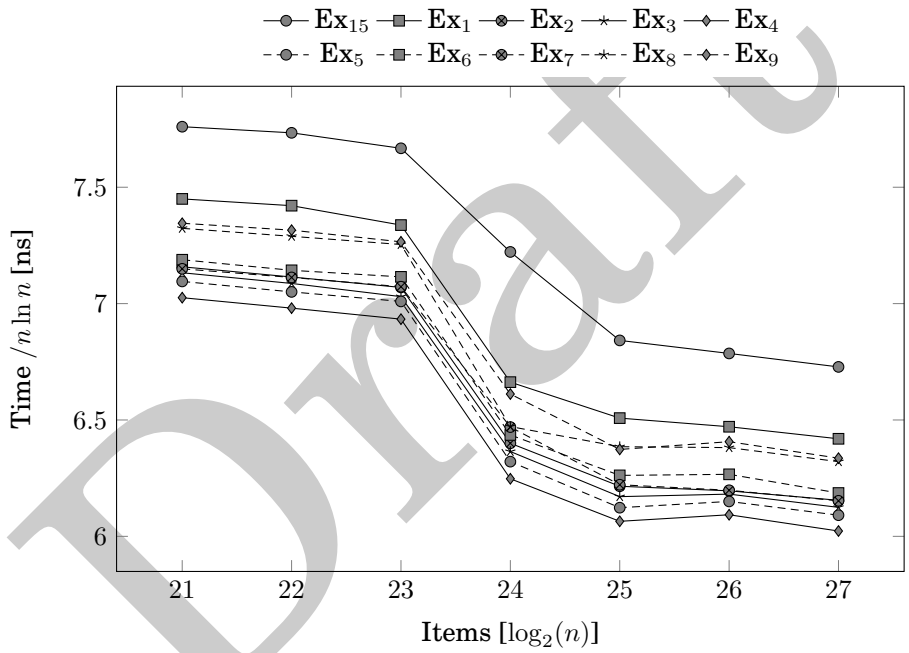

Fig. 16. Running time experiments for k-pivot quicksort algorithms based on the "Exchange$_k$" partitioning strategy. Each data point is the average over 600 trials. Times are scaled by $n \ln n$.

fig:running:times:k: