Algorithmica manuscript No.
(will be inserted by the editor)

Analysis of Pivot Sampling in Dual-Pivot Quicksort

A Holistic Analysis of Yaroslavskiy's Partitioning Scheme

Markus E. Nebel · Sebastian Wild · Conrado Martínez

Received: date / Accepted: date

Abstract The new dual-pivot Quicksort by Vladimir Yaroslavskiy — used in Oracle's Java runtime library since version 7 — features intriguing asymmetries. They make a basic variant of this algorithm use less comparisons than classic single-pivot Quicksort. In this paper, we extend the analysis to the case where the two pivots are chosen as fixed order statistics of a random sample. Surprisingly, dual-pivot Quicksort then needs *more* comparisons than a corresponding version of classic Quicksort, so it is clear that counting comparisons is not sufficient to explain the running time advantages observed for Yaroslavskiy's algorithm in practice. Consequently, we take a more holistic approach and give also the precise leading term of the average number of swaps, the number of executed Java Bytecode instructions and the number of scanned elements, a new simple cost measure that approximates I/O costs in the memory hierarchy. We determine optimal order statistics for each of the cost measures. It turns out that the asymmetries in Yaroslavskiy's algorithm render pivots with a systematic skew more efficient than the symmetric choice. Moreover, we finally have a convincing explanation for the success of Yaroslavskiy's algorithm in practice: Compared with corresponding versions of classic single-pivot Quicksort, dual-pivot Quicksort needs significantly less I/Os, both with and without pivot sampling.

This work has been partially supported by funds from the Spanish Ministry for Economy and Competitiveness (MINECO) and the European Union (FEDER funds) under grant COMMAS (ref. TIN2013-46181-C2-1-R).

A preliminary version of this article was presented at AofA 2014 (Nebel and Wild 2014).

Markus E. Nebel
Computer Science Department, University of Kaiserslautern, Germany; and
Department of Mathematics and Computer Science, University of Southern Denmark, Denmark
E-mail: nebel@cs.uni-kl.de

Sebastian Wild
Computer Science Department, University of Kaiserslautern, Germany, E-mail: wild@cs.uni-kl.de

Conrado Martínez
Department of Computer Science, Univ. Politècnica de Catalunya, E-mail: conrado@cs.upc.edu

Keywords Quicksort, dual-pivot, Yaroslavskiy's partitioning method, median of three, average-case analysis, I/O operations, external-memory model

1 Introduction

Quicksort is one of the most efficient comparison-based sorting algorithms and is thus widely used in practice, for example in the sort implementations of the C++ standard library and Oracle's Java runtime library. Almost all practical implementations are based on the highly tuned version of Bentley and McIlroy (1993), often equipped with the strategy of Musser (1997) to avoid quadratic worst-case behavior. The Java runtime environment was no exception to this — up to version 6. With version 7 released in 2009, Oracle broke with this tradition and replaced its tried and tested implementation by a dual-pivot Quicksort with a new partitioning method proposed by Vladimir Yaroslavskiy.

The decision was based on extensive running time experiments that clearly favored the new algorithm. This was particularly remarkable as earlier analyzed dual-pivot variants had not shown any potential for performance gains over classic single-pivot Quicksort (Sedgewick 1975; Hennequin 1991). However, we could show for pivots from fixed array positions (i.e. no sampling) that Yaroslavskiy's asymmetric partitioning method beats classic Quicksort in the comparison model: asymptotically $1.9\,n \ln n$ vs. $2\,n \ln n$ comparisons on average (Wild and Nebel 2012). It is an interesting question how far one can get by exploiting asymmetries in this way. For dual-pivot Quicksort with an *arbitrary* partitioning method, Aumüller and Dietzfelbinger (2013) establish a lower bound of asymptotically $1.8\,n \ln n$ comparisons and they also propose a partitioning method that attains this bound by dynamically switching the order of comparisons depending on the current subproblem.

The savings in comparisons are opposed by a large increase in the number of swaps, so the competition between classic Quicksort and Yaroslavskiy's Quicksort remained open. To settle it, we compared Java implementations of both variants and found that Yaroslavskiy's method executes *more* Java Bytecode instructions on average (Wild et al 2015). A possible explanation why it still shows better running times was recently given by Kushagra et al (2014): Yaroslavskiy's algorithm in total needs fewer passes over the array than classic Quicksort, and is thus more efficient in the *external-memory model*. (We rederive and extend their results in this article.)

Our analyses cited above ignore a very effective strategy in Quicksort: for decades, practical implementations choose their pivots as *median of a random sample* of the input to be more efficient (both in terms of average performance and in making worst cases less likely). Oracle's Java 7 implementation also employs this optimization: it chooses its two pivots as the *tertiles of five* sample elements. This equidistant choice is a plausible generalization, since selecting the median as pivot is known to be optimal for classic Quicksort (Sedgewick 1975; Martínez and Roura 2001).

However, the classic partitioning methods treat elements smaller and larger than the pivot in symmetric ways — unlike Yaroslavskiy's partitioning algorithm: depending on how elements relate to the two pivots, one of *five* different execution paths is taken in the partitioning loop, and these can have highly different costs! How often

each of these five paths is taken depends on the *ranks* of the two pivots, which we can push in a certain direction by selecting *skewed* order statistics of a sample instead of the tertiles. The partitioning costs alone are then minimized if the cheapest execution path is taken all the time. This however leads to very unbalanced distributions of sizes for the recursive calls, such that a *trade-off* between partitioning costs and balance of subproblem sizes has to be found.

We have demonstrated experimentally that there is potential to tune dual-pivot Quicksort using skewed pivots (Wild et al 2013), but only considered a small part of the parameter space. It will be the purpose of this paper to identify the optimal way to sample pivots by means of a precise analysis of the resulting overall costs, and to validate (and extend) the empirical findings that way.

There are scenarios where, even for the symmetric, classic Quicksort, a skewed pivot can yield benefits over median of k (Martínez and Roura 2001; Kaligosi and Sanders 2006). An important difference to Yaroslavskiy's algorithm is, however, that the situation remains symmetric: a relative pivot rank $\alpha < \frac{1}{2}$ has the same effect as one with rank $1 - \alpha$.

Furthermore, it turns out that dual-pivot Quicksort needs more comparisons than classic Quicksort, if both choose their pivots from a sample (of the same size), but the running time advantages of Yaroslavskiy's algorithm remain, so key comparisons do not dominate running time in practice. As a consequence, we consider other cost measures like the number of executed Bytecode instructions and I/O operations.

1.1 Cost Measures for Sorting

As outlined above, we started our attempt to explain the success of Yaroslavskiy's algorithm by counting comparisons and swaps, as it is classically done for the evaluation of sorting strategies. Since the results were not conclusive, we switched to primitive instructions and determined the expected number of *Java Bytecodes* as well as the number of operations executed by Knuth's *MMIX* computer (see (Wild 2012)), comparing the different Quicksort variants on this basis. To our surprise, Yaroslavskiy's algorithm is not superior in terms of primitive instructions, either.

At this point we were convinced that features of modern computers like memory hierarchies and/or pipelined execution must be responsible for the speedup empirically observed for the new dual-pivot Quicksort. The memory access pattern of partitioning in Quicksort is essentially like for a sequential scan, only that several scans with separate index variables are interleaved: two indices that alternatingly run towards each other in classic Quicksort, the three indices k, g and ℓ in Yaroslavskiy's Quicksort (see Section 3.2) or even four indices in the three-pivot Quicksort of Kushagra et al (2014). We claim that a good cost measure is the *total distance covered by all scanning indices*, which we call the number of *"scanned elements"* (where the number of visited elements is used as the unit of "distance").

As we will show, this cost measure is rather easy to analyze, but it might seem artificial at first sight. It is however closely related to the number of cache misses in practice (see Section 7.2) and the number of I/O operations in the *external-memory model*: For large inputs in external memory, one has to assume that each block of

elements of the input array is responsible for one I/O when it is accessed for the first time in a partitioning run. No spatial locality between accesses through different scanning indices can be assumed, so memory accesses of one index will not save (many) I/Os for another index. Finally, accesses from different partitioning runs lack temporal locality, so (most) elements accessed in previous partitioning runs will have been removed from internal memory before recursively sorting subarrays. Therefore, the number of I/Os is very close to the number of scanned elements, when the blocks contain just single array elements. This is in fact not far from reality for the caches close to the CPU: the L1 and L2 caches in the AMD Opteron architecture, for example, use block sizes of 64 bytes, which on a 64-bit computer means that only 8 array entries fit in one block (Hennessy and Patterson 2006).

The external-memory model is an idealized view itself. Actual hardware has a hierarchy of caches with different characteristics, and for caches near the CPU, only very simple addressing and replacement strategies yield acceptable access delays. From that perspective, we now have three layers of abstraction: Scanned elements are an approximation of I/O operations of the external-memory model (for scanning-based algorithms like Quicksort), which in turn are an approximation of memory hierarchy delays like cache misses.

The theoretical cost measure "scanned elements" has been used implicitly in earlier analyses of the caching behavior of Quicksort and other scanning-based algorithms like, e. g., Mergesort (LaMarca and Ladner 1999; Kushagra et al 2014), even though it has (to our knowledge) never been made explicit; it was merely used as an intermediate step of the analysis. In particular, Kushagra et al essentially compute the number of scanned elements for different Quicksort variants for the case of random pivots (i. e., no sampling), and find that Yaroslavskiy's algorithm outperforms classic Quicksort in this cost measure.

Besides the memory hierarchy, the effects of pipelined execution might be an explanation for the speedup observed for the new algorithm. However, the numbers of branch misses (a. k. a. pipeline stalls) incurred by classic Quicksort and Yaroslavskiy's Quicksort do not differ significantly under simple branch predictions schemes (Martínez et al 2015), so pipelining is not a convincing explanation.

The rest of this article is organized as follows: After listing some general notation, Section 3 introduces the subject of study: Yaroslavskiy's algorithm. Section 4 collects the main analytical results of this paper, the proof of which is given in Sections 5 and 6. Mathematical arguments in the main text are kept concise, but the interested reader is provided with details in the appendices. In Section 7, we compare the analytical result with experimental data for practical input sizes. The algorithmic consequences of our analysis are discussed in Section 8 in detail. Section 9 concludes the paper.

2 Notation and Preliminaries

We write vectors in bold font, for example $\mathbf{t} = (t_1, t_2, t_3)$. For concise notation, we use expressions like $\mathbf{t} + 1$ to mean *element-wise* application, i.e., $\mathbf{t} + 1 = (t_1 + 1, t_2 + 1, t_3 + 1)$. By $\mathrm{Dir}(\boldsymbol{\alpha})$, we denote a random variable with *Dirichlet distribution* and shape parameter $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_d) \in \mathbb{R}^d_{>0}$. Likewise for parameters $n \in \mathbb{N}$

and $\mathbf{p} = (p_1, \dots, p_d) \in [0,1]^d$ with $p_1 + \cdots + p_d = 1$, we write $\mathrm{Mult}(n, \mathbf{p})$ for a random variable with *multinomial distribution* with n trials. $\mathrm{HypG}(k, r, n)$ is a random variable with *hypergeometric distribution*, i.e., the number of red balls when drawing k times without replacement from an urn of $n \in \mathbb{N}$ balls, r of which are red, (where $k, r \in \{1, \dots, n\}$). Finally, $\mathcal{U}(a, b)$ is a random variable uniformly distributed in the interval (a, b), and $\mathrm{B}(p)$ is a Bernoulli variable with probability p to be 1. We use "$\overset{\mathcal{D}}{=}$" to denote equality in distribution.

As usual for the average case analysis of sorting algorithms, we assume the *random permutation model*, i.e., all elements are different and every ordering of them is equally likely. The input is given as an array $\mathbf{A}$ of length n and we denote the initial entries of $\mathbf{A}$ by $U_1, \dots, U_n$. We further assume that $U_1, \dots, U_n$ are i. i. d. uniformly $\mathcal{U}(0, 1)$ distributed; as their ordering forms a random permutation (Mahmoud 2000), this assumption is without loss of generality. Some further notation specific to our analysis is introduced below; for reference, we summarize all notations used in this paper in Appendix A.

3 Generalized Yaroslavskiy Quicksort

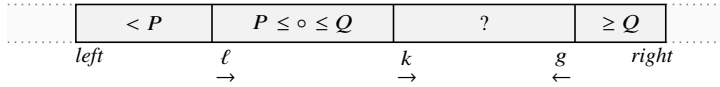
In this section, we review Yaroslavskiy's partitioning method and combine it with the pivot sampling optimization to obtain what we call the *Generalized Yaroslavskiy Quicksort* algorithm. We give a full-detail implementation of the algorithm, because *preservation of randomness* is somewhat tricky to achieve in presence of pivot sampling, but vital for precise analysis. The code we give here can be fully analyzed, but is admittedly not suitable for productive use; it should rather be considered as a mathematical *model* for practical implementations, which often do *not* preserve randomness (see, e. g., the discussion of Java 7's implementation below).

3.1 Generalized Pivot Sampling

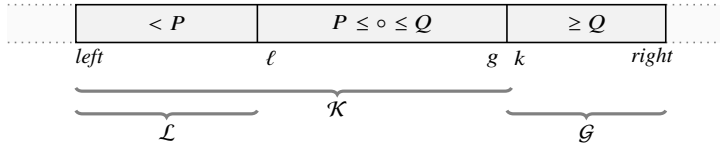
Our pivot selection process is declaratively specified as follows, where $\mathbf{t} = (t_1, t_2, t_3) \in \mathbb{N}^3$ is a fixed parameter: Choose a random sample $\mathbf{V} = (V_1, \dots, V_k)$ of size $k = k(\mathbf{t}) \coloneqq t_1 + t_2 + t_3 + 2$ from the elements and denote by $(V_{(1)}, \dots, V_{(k)})$ the *sorted* sample, i.e., $V_{(1)} \le V_{(2)} \le \cdots \le V_{(k)}$. (In case of equal elements any possible ordering will do; in this paper, we assume distinct elements.) Then choose the two pivots $P \coloneqq V_{(t_1+1)}$ and $Q \coloneqq V_{(t_1+t_2+2)}$ such that they divide the sorted sample into three regions of respective sizes t_1, t_2 and t_3:

$$\underbrace{V_{(1)} \dots V_{(t_1)}}_{t_1 \text{ elements}} \le \underbrace{V_{(t_1+1)}}_{=P} \le \underbrace{V_{(t_1+2)} \dots V_{(t_1+t_2+1)}}_{t_2 \text{ elements}} \le \underbrace{V_{(t_1+t_2+2)}}_{=Q} \le \underbrace{V_{(t_1+t_2+3)} \dots V_{(k)}}_{t_3 \text{ elements}}.$$

The parameter choice $\mathbf{t} = (0, 0, 0)$ corresponds to the case without sampling. Note that by definition, P is the small(er) pivot and Q is the large(r) one. We refer to the $k - 2$ elements of the sample that are not chosen as pivots as *"sampled-out"*; P and Q are the chosen *pivots*. All other elements — those which have not been part of the sample — are referred to as *ordinary* elements.



(a) Invariant of Algorithm 1 during partitioning.



(b) State after the partitioning loop has been left.

Figure 1 The state of the array $\mathtt{A}$ during and after partitioning. Note that the last values attained by k, g and ℓ are *not* used to access the array $\mathtt{A}$, so the positions of the indices after partitioning are by definition not contained in the corresponding position sets.

We assume that the sample size k does not depend on the size n of the current (sub)problem for several reasons: First of all, such strategies are not very practical because they complicate code. Furthermore, if the sample size grows recognizably with n, they need a sorting method for the samples that is efficient also when samples get large. If, on the other hand, k grows very slowly with n, the sample is essentially constant for practical input sizes.

Analytically, any growing sample size $k = k(n) = \omega(1)$ immediately provides asymptotically *precise* order statistics (*law of large numbers*) and thus allows an optimal choice of the pivots. As a consequence, the leading term of costs is the *same* for all such sample sizes and only the linear term of costs is affected (as long as $k = O(n^{1-\epsilon})$), see Martínez and Roura (2001). This would make it impossible to distinguish pivot selection strategies by looking at leading-term asymptotics.

Note that with $k = O(1)$, we hide the cost of selecting order statistics in the second order term, so our leading-term asymptotics ignores the costs of sorting the sample in the end. However, it is a fixed constant whose contribution we can still roughly estimate (as validated in Section 7). Also, we retain the possibility of letting $k \to \infty$ analytically (see Section 8.3).

3.2 Yaroslavskiy's Dual-Pivot Partitioning Method

Yaroslavskiy's partitioning method is given in Algorithm 1. In bird's-eye view, it consists of two indices, k and g, that start at the left resp. right end of $\mathtt{A}$ and scan the array until they meet. Elements left of k are smaller or equal than Q, elements right of g are larger. Additionally, a third index ℓ lags behind k and separates elements smaller than P from those between both pivots. Graphically speaking, this invariant of the algorithm is given in Figure 1(a).

When partitioning is finished, k and g have met and thus ℓ and g divide the array into three ranges; precisely speaking, in line 23 of Algorithm 1 the array has the shape shown in Figure 1(b).

Algorithm 1 Yaroslavskiy's dual-pivot partitioning algorithm.

PartitionYaroslavskiy ($\mathbf{A}$, *left*, *right*, P, Q)

// Assumes *left* $\leq$ *right*.
// Rearranges $\mathbf{A}$ s. t. with return value (i_p, i_q) holds $\begin{cases} \mathbf{A}[j] < P, & \text{for } left \leq j \leq i_p; \\ P \leq \mathbf{A}[j] \leq Q, & \text{for } i_p < j < i_q; \\ \mathbf{A}[j] \geq Q, & \text{for } i_q \leq j \leq right. \end{cases}$

```
 1  ℓ := left;   g := right;   k := ℓ
 2  while k ≤ g
 3      if A[k] < P
 4          Swap A[k] and A[ℓ]
 5          ℓ := ℓ + 1
 6      else
 7          if A[k] ≥ Q
 8              while A[g] > Q and k < g
 9                  g := g − 1
10              end while
11              if A[g] ≥ P
12                  Swap A[k] and A[g]
13              else
14                  Swap A[k] and A[g]
15                  Swap A[k] and A[ℓ]
16                  ℓ := ℓ + 1
17              end if
18              g := g − 1
19          end if
20      end if
21      k := k + 1
22  end while
23  return (ℓ − 1, g + 1)
```

We write $\mathcal{K}$, $\mathcal{G}$ and $\mathcal{L}$ for the sets of all indices that k, g resp. ℓ attain in the course of the partitioning process — more precisely: $\mathcal{K}$ is the set of all values attained by variable k, for which we access the array via $\mathbf{A}[k]$; similarly for $\mathcal{G}$ and $\mathcal{L}$. (We need a precise definition for the analysis later.[1]) As the indices move sequentially these sets are in fact (integer) intervals, as indicated in Figure 1(b).

Moreover, we call an element *small*, *medium*, or *large* if it is smaller than P, between P and Q, or larger than Q, respectively. The following properties of the algorithm are needed for the analysis, (see Wild and Nebel (2012); Wild et al (2015) for details):

(Y1) Elements U_i with $i \in \mathcal{K}$ are first compared with P (line 3). Only if U_i is not small, it is also compared to Q (line 7).

(Y2) Elements U_i with $i \in \mathcal{G}$ are first compared with Q (line 8). If they are not large, they are also compared to P (line 11).

(Y3) Every small element $U_i < P$ eventually causes one swap to put it behind ℓ (at line 4 if $i \in \mathcal{K}$ resp. at line 15 if $i \in \mathcal{G}$).

[1] Note that the meaning of $\mathcal{L}$ is different in our previous work (Wild et al 2015): therein $\mathcal{L}$ includes the last value index variable ℓ attains which is never used to access the array. The authors consider the new definition clearer and therefore decided to change it.

(Y4) The large elements located in $\mathcal{K}$ and the non-large elements in $\mathcal{G}$ are always swapped in pairs (line 12 resp. line 14).

For the number of comparisons we will (among other quantities) need to count the large elements $U_i > Q$ with $i \in \mathcal{K}$, cf. (Y1). We abbreviate their number by "$l@\mathcal{K}$". Similarly, $s@\mathcal{K}$ and $s@\mathcal{G}$ denote the number of small elements in k's resp. g's range.

3.3 Implementing Generalized Pivot Sampling

While extensive literature on the analysis of (single-pivot) Quicksort with pivot sampling is available, most works do not specify the pivot selection process in detail. (Noteworthy exceptions are Sedgewick's seminal works which give detailed code for the median-of-three strategy (Sedgewick 1975, 1978) and Bentley and McIlroy's influential paper on engineering a practical sorting method (Bentley and McIlroy 1993).) The usual justification is that, in any case, we only draw pivots a *linear* number of times and from a constant-size sample. So the costs of pivot selection are negligible for the leading-term asymptotic, and hence also the precise way of how selection is done is not important.

There is one caveat in the argumentation: Analyses of Quicksort usually rely on setting up a recurrence equation of expected costs that is then solved (precisely or asymptotically). This in turn requires the algorithm to *preserve* the distribution of input permutations for the subproblems subjected to recursive calls — otherwise the recurrence does not hold. Most partitioning algorithms, including the one of Yaroslavskiy, have the desirable property to preserve randomness (Wild and Nebel 2012); but this is not sufficient! We also have to make sure that the main procedure of Quicksort does not alter the distribution of inputs for recursive calls; in connection with elaborate pivot sampling algorithms, this is harder to achieve than it might seem at first sight.

For these reasons, the authors felt the urge to include a minute discussion of how to implement the generalized pivot sampling scheme of Section 3.1 in such a way that the recurrence equation remains precise. We have to address the following questions:

Which elements to choose for the sample? In theory, a *random* sample produces the most reliable results and also protects against worst case inputs. The use of a random pivot for classic Quicksort has been considered right from its invention (Hoare 1961) and is suggested as a general strategy to deal with biased data (Sedgewick 1978).

However, all programming libraries known to the authors actually avoid the additional effort of drawing random samples. They use a set of deterministically selected positions of the array, instead; chosen to give reasonable results for common special cases like almost sorted arrays. For example, the positions used in Oracle's Java 7 implementation are depicted in Figure 2.

For our analysis, the input consists of i. i. d. random variables, so *all* subsets (of a certain size) have the same distribution. We might hence select the positions of sample elements such that they are convenient for our (analysis) purposes. For reasons elaborated in Section 3.4 below, we have to *exclude* sampled-out elements from partitioning to keep analysis feasible, and therefore, our implementation uses

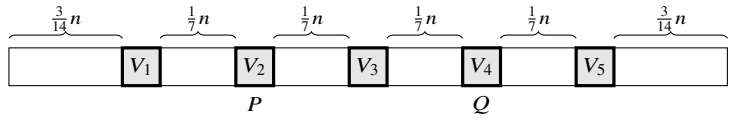


Figure 2 The five sample elements in Oracle's Java 7 implementation of Yaroslavskiy's dual-pivot Quicksort are chosen such that their distances are approximately as given above.

the $t_1 + t_2 + 1$ leftmost and the $t_3 + 1$ rightmost elements of the array as sample, as illustrated in Figure 3. Then, partitioning can simply be restricted to the range between the two parts of the sample, namely positions $t_1 + t_2 + 2$ through $n - t_3 - 1$ (cf. line 17 of Algorithm 2).

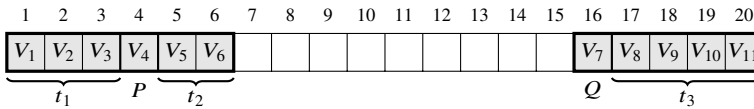


Figure 3 Location of the sample in our implementation of $\mathrm{YQS}^w_{\mathbf{t}}$ with $\mathbf{t} = (3, 2, 4)$. Only the non-shaded region $\mathbf{A}[7..15]$ is subject to partitioning.

How do we select the desired order statistics from the sample? Finding a given order statistic of a list of elements is known as the *selection problem* and can be solved by specialized algorithms like Quickselect. Even though these selection algorithms are superior by far on large lists, selecting pivots from a reasonably small sample is most efficiently done by fully sorting the whole sample with an elementary sorting method. Once the sample has been sorted, we find the pivots in $\mathbf{A}[t_1 + 1]$ and $\mathbf{A}[n - t_3]$, respectively.

We will use an Insertionsort variant for sorting samples. Note that the implementation has to "jump" across the gap between the left part and the right part of the sample. Algorithm 5 (page 12) and its symmetric cousin Algorithm 6 do that by internally ignoring the gap in index variables and then correct for that whenever the array is actually accessed.

How do we deal with sampled-out elements? As discussed in Section 3.4, we exclude sampled-out elements from the partitioning range. After partitioning, we thus have to move the t_2 sampled-out elements, which actually belong between the pivots, to the middle partition. Moreover, the pivots themselves have to be swapped in place. This process is illustrated in Figure 4 and spelled out in lines 18 – 21 of Algorithm 2. Note that the order of swaps has been chosen carefully to correctly deal with cases where the regions to be exchanged overlap.

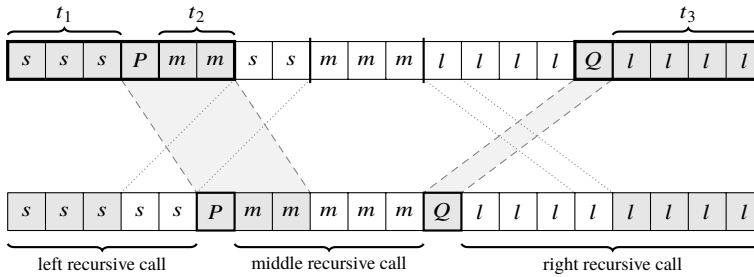


Figure 4 First row: State of the array just after partitioning the ordinary elements (after line 17 of Algorithm 2). The letters indicate whether the element at this location is smaller (s), between (m) or larger (l) than the two pivots P and Q. Sample elements are shaded.
Second row: State of the array after pivots and sample parts have been moved to their partition (after line 21). The "rubber bands" indicate moved regions of the array.

3.4 Randomness Preservation

For analysis, it is vital to preserve the input distribution for recursive calls, as this allows us to set up a recurrence equation for costs. While Yaroslavskiy's method (as given in Algorithm 1) preserves randomness inside partitions, pivot sampling requires special care. For efficiently selecting the pivots, we *sort* the entire sample, so the sampled-out elements are far from randomly ordered; including them in partitioning would not produce randomly ordered subarrays! But there is also no need to include them in partitioning, as we already have the sample divided into the three groups of t_1 small, t_2 medium and t_3 large elements. All ordinary elements are still in random order and Yaroslavskiy's partitioning divides them into three randomly ordered subarrays.

What remains problematic is the order of elements for recursive calls. The second row in Figure 4 shows the situation after all sample elements (shaded gray) have been put into the correct subarray. As the sample was sorted, the left and middle subarrays have sorted prefixes of length t_1 resp. t_2 followed by a random permutation of the remaining elements. Similarly, the right subarray has a sorted suffix of t_3 elements. So the subarrays are *not* randomly ordered, (except for the trivial case $\mathbf{t} = 0$)! How shall we deal with this non-randomness?

The maybe surprising answer is that we can indeed *exploit* this non-randomness; not only in terms of a precise analysis, but also for efficiency: the sorted part *always* lies completely inside the *sample range* for the next partitioning phase. So our specific kind of non-randomness only affects sorting the sample (in subsequent recursive calls), but it never affects the partitioning process itself!

It seems natural that sorting should somehow be able to profit from partially sorted input, and in fact, many sorting methods are known to be *adaptive* to existing order (Estivill-Castro and Wood 1992). For our special case of a fully sorted prefix or suffix of length $s \geq 1$ and a fully random rest, we can simply use Insertionsort where the first s iterations of the outer loop are skipped. Our Insertionsort implementations accept s as an additional parameter.

For Insertionsort, we can also precisely *quantify* the savings resulting from skipping the first s iterations: Apart from per-call overhead, we save exactly what it would

Algorithm 2 Yaroslavskiy's Dual-Pivot Quicksort with Generalized Pivot Sampling

```
GeneralizedYaroslavskiy (A, left, right, type)
    // Assumes left ≤ right, w ≥ k − 1
    // Sorts A[left, ..., right].
 1  if right − left < w
 2      case distinction on type
 3          in case root   do InsertionSortLeft  (A, left, right, 1)
 4          in case left   do InsertionSortLeft  (A, left, right, max{t1, 1})
 5          in case middle do InsertionSortLeft  (A, left, right, max{t2, 1})
 6          in case right  do InsertionSortRight (A, left, right, max{t3, 1})
 7      end cases
 8  else
 9      case distinction on type    // Sort sample
10          in case root   do SampleSortLeft  (A, left, right, 1)
11          in case left   do SampleSortLeft  (A, left, right, max{t1, 1})
12          in case middle do SampleSortLeft  (A, left, right, max{t2, 1})
13          in case right  do SampleSortRight (A, left, right, max{t3, 1})
14      end cases
15      p := A[left + t1];   q := A[right − t3]
16      partLeft := left + t1 + t2 + 1;   partRight := right − t3 − 1
17      (ip, iq) := PartitionYaroslavskiy (A, partLeft, partRight, p, q)
        // Swap middle part of sample and p to final place (cf. Figure 4)
18      for j := t2, ..., 0    // iterate downwards
19          Swap A[left + t1 + j] and A[ip − t2 + j]
20      end for
        // Swap q to final place.
21      Swap A[iq] and A[partRight + 1]
22      GeneralizedYaroslavskiy (A, left,          ip − t2 − 1, left  )
23      GeneralizedYaroslavskiy (A, ip − t2 + 1, iq − 1,        middle)
24      GeneralizedYaroslavskiy (A, iq + 1,        right,       right )
25  end if
```

Algorithm 3 Insertionsort "from the left", exploits sorted prefixes.

```
InsertionSortLeft(A, left, right, s)
    // Assumes left ≤ right and s ≤ right − left − 1.
    // Sorts A[left, ..., right], assuming that the s leftmost elements are already sorted.
1  for i = left + s, ..., right
2      j := i − 1;   v := A[i]
3      while j ≥ left ∧ v < A[j]
4          A[j + 1] := A[j];   j := j − 1
5      end while
6      A[j + 1] := v
7  end for
```

have costed to sort a random permutation of the length of this prefix/suffix with Insertionsort. As all prefixes/suffixes have constant lengths (independent of the length of the current subarray), precise analysis remains feasible, see Section 5.1.

Algorithm 4 Insertionsort "from the right", exploits sorted suffixes.

InsertionSortRight(A, *left*, *right*, *s*)

// Assumes $left \leq right$ and $s \leq right - left - 1$.
// Sorts A[*left*, . . . , *right*], assuming that the *s rightmost* elements are already sorted.

```
1  for i = right − s , . . . , left    // iterate downwards
2      j := i + 1;    v := A[i]
3      while j ≤ right ∧ v > A[j]
4          A[j − 1] := A[j];    j := j + 1
5      end while
6      A[j − 1] := v
7  end for
```

Algorithm 5 Sorts the sample with Insertionsort "from the left"

SampleSortLeft(A, *left*, *right*, *s*)

// Assumes $right - left + 1 \geq k$ and $s \leq t_1 + t_2 + 1$.
// Sorts the k elements A[*left*], . . . , A[$left + t_1 + t_2$], A[$right - t_3$], . . . , A[*right*],
// assuming that the s leftmost elements are already sorted.

// A[[i]] is used as abbreviation for A[$i + offset$], where *offset* has to be computed as follows:
// **if** $i > left + t_1 + t_2$ **then** $offset := n - k$ **else** $offset := 0$ **end if**,
// where $n = right - left + 1$.

```
1  InsertionSortLeft(A, left, left + t1 + t2, s)
2  for i = left + t1 + t2 + 1 , . . . , left + k − 1
3      j := i − 1;    v := A[[i]]
4      while j ≥ left ∧ v < A[[j]]
5          A[[j + 1]] := A[[j]];    j := j − 1
6      end while
7      A[[j + 1]] := v
8  end for
```

Algorithm 6 Sorts the sample with Insertionsort "from the right"

SampleSortRight(A, *left*, *right*, *s*)

// Assumes $right - left + 1 \geq k$ and $s \leq t_3 + 1$.
// Sorts the k elements A[*left*], . . . , A[$left + t_1 + t_2$], A[$right - t_3$], . . . , A[*right*],
// assuming that the s rightmost elements are already sorted.

// A[[i]] is used as abbreviation for A[$i + offset$], where *offset* has to be computed as follows:
// **if** $i > left + t_1 + t_2$ **then** $offset := n - k$ **else** $offset := 0$ **end if**,
// where $n = right - left + 1$.

```
1  InsertionSortRight(A, right − t3, right, s)
2  for i = left + k − t3 − 2, . . . , left    // iterate downwards
3      j := i + 1;    v := A[[i]]
4      while j ≤ left + k − 1 ∧ v > A[[j]]
5          A[[j − 1]] := A[[j]];    j := j + 1
6      end while
7      A[[j − 1]] := v
8  end for
```

3.5 Generalized Yaroslavskiy Quicksort

Combining the implementation of generalized pivot sampling — paying attention to the subtleties discussed in the previous sections — with Yaroslavskiy's partitioning method, we finally obtain Algorithm 2. We refer to this sorting method as *Generalized Yaroslavskiy Quicksort* with pivot sampling parameter $\mathbf{t} = (t_1, t_2, t_3)$ and Insertionsort threshold w, shortly written as $\mathrm{YQS}^w_{\mathbf{t}}$. We assume that $w \geq k - 1 = t_1 + t_2 + t_3 + 1$ to make sure that every partitioning step has enough elements for pivot sampling.

The last parameter of Algorithm 2 tells the current call whether it is a topmost call (`root`) or a recursive call on a left, middle or right subarray of some earlier invocation. By that, we know which part of the array is already sorted: for `root` calls, we cannot rely on anything being sorted, in `left` and `middle` calls, we have a sorted prefix of length t_1 resp. t_2, and for a `right` call, the t_3 rightmost elements are known to be in order. The initial call then takes the form GeneralizedYaroslavskiy ($\mathbf{A}$, 1, n, `root`).

4 Results

For $\mathbf{t} \in \mathbb{N}^3$ and $H_n = \sum_{i=1}^{n} \frac{1}{i}$ the nth harmonic number, we define the *discrete entropy* $\mathcal{H} = \mathcal{H}(\mathbf{t})$ of $\mathbf{t}$ as

$$\mathcal{H}(\mathbf{t}) \;=\; \sum_{r=1}^{3} \frac{t_r + 1}{k+1} (H_{k+1} - H_{t_r+1}) \,. \tag{1}$$

The name is justified by the following connection between $\mathcal{H}$ and the *entropy function* $\mathcal{H}^*$ of information theory: for the sake of analysis, let $k \to \infty$, such that ratios t_r/k converge to constants τ_r. Then

$$\mathcal{H} \;\sim\; -\sum_{r=1}^{3} \tau_r (\ln(t_r + 1) - \ln(k+1)) \;\sim\; -\sum_{r=1}^{3} \tau_r \ln(\tau_r) \;=:\; \mathcal{H}^*(\boldsymbol{\tau}) \,. \tag{2}$$

The first step follows from the asymptotic equivalence $H_n \sim \ln(n)$ as $n \to \infty$. Equation (2) shows that for large $\mathbf{t}$, the maximum of $\mathcal{H}$ is attained for $\tau_1 = \tau_2 = \tau_3 = \frac{1}{3}$. Now we state our main result.

Theorem 4.1 (Main theorem): *Generalized Yaroslavskiy Quicksort with pivot sampling parameter $\mathbf{t} = (t_1, t_2, t_3)$ performs on average $C_n \sim \frac{a_C}{\mathcal{H}} n \ln n$ comparisons, $S_n \sim \frac{a_S}{\mathcal{H}} n \ln n$ swaps and $SE_n \sim \frac{a_{SE}}{\mathcal{H}} n \ln n$ element scans to sort a random permutation of n elements, where*

$$\begin{aligned} a_C &= 1 + \frac{t_2 + 1}{k+1} + \frac{(2t_1 + t_2 + 3)(t_3 + 1)}{(k+1)(k+2)}, \\ a_S &= \frac{t_1 + 1}{k+1} + \frac{(t_1 + t_2 + 2)(t_3 + 1)}{(k+1)(k+2)} \qquad \textit{and} \\ a_{SE} &= 1 + \frac{t_1 + 1}{k+1} \,. \end{aligned}$$

Moreover, if the partitioning loop is implemented as in Appendix C of (*Wild et al* 2015)*, it executes on average* $BC_n \sim \frac{a_{BC}}{\mathcal{H}}\, n \ln n$ *Java Bytecode instructions to sort a random permutation of size* n *with*

$$a_{BC} = 10 + 13\frac{t_1+1}{k+1} + 5\frac{t_2+1}{k+1} + 11\frac{(t_1+t_2+2)(t_3+1)}{(k+1)(k+2)} + \frac{(t_1+1)(t_1+t_2+3)}{(k+1)(k+2)}\,.$$

The following sections are devoted to the proof of Theorem 4.1. Section 5 sets up a recurrence of costs and characterizes the distribution of costs of one partitioning step. The expected values of the latter are computed in Section 6.1. Finally, Section 6.2 provides a generic solution to the recurrence of the expected costs; in combination with the expected partitioning costs, this concludes our proof.

5 Distributional Analysis

5.1 Recurrence Equations of Costs

Let us denote by $C_n^{\mathtt{root}}$ the *costs* of $\mathrm{YQS}_{\mathbf{t}}^{w}$ on a random permutation of size n — where the different *cost measures* introduced in Section 1.1 will take the place of $C_n^{\mathtt{root}}$ later. $C_n^{\mathtt{root}}$ is a non-negative *random* variable whose distribution depends on n. The total costs decompose into those for the first partitioning step plus the costs for recursively solving subproblems.

Due to our implementation of the pivot sampling method (see Section 3.3), the costs for a recursive call do not only depend on the size of the subarray, but also on the *type* of the call, i.e., whether it is a left, middle or right subproblem or the topmost call: Depending on the type, a part of the array will already be in order, which we exploit either in sorting the sample (if $n > w$) or in sorting the whole subarray by Insertionsort (if $n \le w$). We thus write $C_n^{\mathtt{type}}$ for the (random) cost of a call to GeneralizedYaroslavskiy($\mathbf{A}, i, j, \mathtt{type}$) with $j - i - 1 = n$ (i.e., $\mathbf{A}[i..j]$ contains n elements) where `type` can either be `root` (for the initial topmost call) or one of `left`, `middle` and `right`.

As Yaroslavskiy's partitioning method applied to a random permutation always generates subproblems with the same distribution (see Section 3.4), we can express the total costs recursively in terms of the same cost functions with smaller arguments: for sizes J_1, J_2 and J_3 of the three subproblems, the costs of corresponding recursive calls are distributed like $C_{J_1}^{\mathtt{left}}$, $C_{J_2}^{\mathtt{middle}}$ and $C_{J_3}^{\mathtt{right}}$, and conditioned on $\mathbf{J} = (J_1, J_2, J_3)$, these random variables are independent. Note, however, that the subproblem sizes are themselves random and not independent of each other (they have to sum to $n-2$). Denoting by $T_n^{\mathtt{type}}$ the (random) cost contribution of the first partitioning round to $C_n^{\mathtt{type}}$, we obtain the following *distributional recurrence* for the four families $(C_n^{\mathtt{type}})_{n\in\mathbb{N}}$ of random variables with $\mathtt{type} \in \{\mathtt{root}, \mathtt{left}, \mathtt{middle}, \mathtt{right}\}$:

$$C_n^{\mathtt{type}} \overset{\mathcal{D}}{=} \begin{cases} T_n^{\mathtt{type}} + C_{J_1}^{\mathtt{left}} + C_{J_2}^{\mathtt{middle}} + C_{J_3}^{\mathtt{right}}, & \text{for } n > w; \\ W_n^{\mathtt{type}}, & \text{for } n \le w. \end{cases} \tag{3}$$

Here $W_n^{\texttt{type}}$ denotes the (random) cost of sorting a subarray of size $n \le w$ using Insertionsort from a (recursive) call of type `type`. We call $T_n^{\texttt{type}}$ the *toll functions* of the recurrence, as they quantify the "toll" we have to pay for unfolding the recurrence once. Our cost measures only differ in the toll functions, such that we can treat them all in a uniform fashion by studying Equation (3).

Dealing with the mutually recursive quantities of Equation (3) is rather inconvenient, but we can luckily avoid it for our purposes. $T_n^{\texttt{root}}$, $T_n^{\texttt{left}}$, $T_n^{\texttt{middle}}$ and $T_n^{\texttt{right}}$ (potentially) differ in the cost of selecting pivots from the sample, but they do *not* differ in the cost caused by the partitioning procedure itself: in all four cases, we invoke Partition on a subarray containing $n-k$ elements that are in random order and the (random) pivot values P and Q always have the same distribution. As we assume that the sample size k is a constant independent of n, the toll functions differ by a constant at most; in fact for all `type`s, we have $T_n^{\texttt{type}} \overset{\mathcal{D}}{=} T_n + O(1)$ where T_n denotes the cost caused by Partition alone. Since the total costs are a linear function of the toll costs, we can separately deal with the two summands. The contribution of the $O(1)$ toll to the overall costs is then trivially bounded by $O(n)$, as two (new) elements are chosen as pivots in each partitioning step, so we can have at most $n/2$ pivot sampling rounds in total.

Similarly, $W_n^{\texttt{type}} \overset{\mathcal{D}}{=} W_n + O(1)$, where W_n denotes the (random) costs of sorting a random permutation of size n with Insertionsort (without skipping the first few iterations). The contribution of Insertionsort to the total costs are in $O(n)$ as the Insertionsort threshold w is constant and we can only have a linear number of calls to Insertionsort. So for the leading term, the precise form of W_n is immaterial. In summary, we have shown that $C_n^{\texttt{type}} \overset{\mathcal{D}}{=} C_n + O(n)$, and in particular $C_n^{\texttt{root}} \overset{\mathcal{D}}{=} C_n + O(n)$, where the distribution of C_n is defined by the following distributional recurrence:

$$C_n \overset{\mathcal{D}}{=} \begin{cases} T_n + C_{J_1} + C'_{J_2} + C''_{J_3}, & \text{for } n > w; \\ W_n, & \text{for } n \le w, \end{cases} \tag{4}$$

with $(C'_j)_{j\in\mathbb{N}}$ and $(C''_j)_{j\in\mathbb{N}}$ independent copies of $(C_j)_{j\in\mathbb{N}}$, i.e., for all j, the variables C_j, C'_j and C''_j are identically distributed and for all $\mathbf{j} \in \mathbb{N}^3$, C_{j_1}, C'_{j_2} and C''_{j_3} are (totally) independent[2], and they are also independent of T_n.

To obtain an expression for $\mathbb{P}(\mathbf{J} = \mathbf{j})$, we note that there are $\binom{n}{k}$ ways to choose k out of n given elements in total. If there shall be exactly j_1 small, j_2 medium and j_3 large elements, we have to choose t_1 of the j_1 small elements for the sample, plus t_2 of the j_2 medium and t_3 of the j_3 large elements. Combining all possible ways to do so gives the number of samples that are consistent with subproblem sizes $\mathbf{j} = (j_1, j_2, j_3)$; we thus have

$$\mathbb{P}(\mathbf{J} = \mathbf{j}) = \binom{j_1}{t_1}\binom{j_2}{t_2}\binom{j_3}{t_3} \Big/ \binom{n}{k}. \tag{5}$$

[2] Total independence means that the joint probability function of all random variables factorizes into the product of the individual probability functions (Chung 2001, p. 53), and does so not only pairwise.

Quantity				Distribution given $\mathbf{I}$
δ	=	$\mathbb{1}_{\{U_\chi > Q\}}$	$\overset{\mathcal{D}}{=}$	$\mathrm{B}(\frac{I_3}{n-k})$
$\lvert\mathcal{K}\rvert$	=	$I_1 + I_2 + \delta$	$\overset{\mathcal{D}}{=}$	$I_1 + I_2 + \mathrm{B}(\frac{I_3}{n-k})$
$\lvert\mathcal{G}\rvert$	=	I_3	$\overset{\mathcal{D}}{=}$	I_3
$\lvert\mathcal{L}\rvert$	=	I_1	$\overset{\mathcal{D}}{=}$	I_1
$l@\mathcal{K}$	=	$(l@\mathcal{K}') + \delta$	$\overset{\mathcal{D}}{=}$	$\mathrm{HypG}(I_1 + I_2, I_3, n-k) + \mathrm{B}(\frac{I_3}{n-k})$
$s@\mathcal{G}$			$\overset{\mathcal{D}}{=}$	$\mathrm{HypG}(I_3, I_1, n-k)$

Table 1 Quantities that arise in the analysis of Partition (Algorithm 1) and their distribution conditional on $\mathbf{I}$. A detailed discussion of these quantities and their distributions is given in (Wild et al 2015).
Note that $\lvert\mathcal{K}\rvert$ depends on δ, which is inconvenient for further analysis, so we work with $\mathcal{K}'$, defined as the first $I_1 + I_2$ elements of $\mathcal{K}$. When $\delta = 0$ we have $\mathcal{K}' = \mathcal{K}$, see (Wild et al 2015) for details.

5.2 Distribution of Partitioning Costs

Recall that we only have to partition the *ordinary* elements, i.e., the elements that have *not* been part of the sample (cf. line 17 of Algorithm 2). Let us denote by I_1, I_2 and I_3 the number of small, medium and large elements among these elements, i.e., $I_1 + I_2 + I_3 = n - k$. Stated differently, $\mathbf{I} = (I_1, I_2, I_3)$ is the vector of sizes of the three partitions (excluding sampled-out elements). There is a close relation between the vectors of *partition* sizes $\mathbf{I}$ and *subproblem* sizes $\mathbf{J}$; we only have to add the sampled-out elements again before the recursive calls: $\mathbf{J} = \mathbf{I} + \mathbf{t}$ (see Figure 4).

Moreover, we define the indicator variable $\delta = \mathbb{1}_{\{U_\chi > Q\}}$ where χ is the array position on which indices k and g first meet. δ is needed to account for an idiosyncrasy of Yaroslavskiy's algorithm: depending on the element U_χ that is initially located at the position where k and g first meet, k overshoots g at the end by either 2 — namely if $U_\chi > Q$ — or by 1, otherwise (Wild et al 2015, "Crossing-Point Lemma").

As we will see, we can precisely characterize the distribution of partitioning costs *conditional* on $\mathbf{I}$, i.e., when considering $\mathbf{I}$ *fixed*. Therefore, we give the conditional distributions of all quantities relevant for the analysis in Table 1. They essentially follow directly from the discussion in our previous work (Wild et al 2015), but for convenience, we give the main arguments again in this paper.

Recall that I_1, I_2 and I_3 are the number of small, medium and large elements, respectively. Since the elements right of g after partitioning are exactly all large elements (see also Figure 1(b)), g scans I_3 elements. Note that the last value that variable g attains is not part of $\mathcal{G}$, since it is never used to access the array.

All small and medium elements are for sure left of k after partitioning. But k might also run over the first large element, if k and g meet on a large element. Therefore, $\lvert\mathcal{K}\rvert = I_1 + I_2 + \delta$ (see also the "Crossing-Point Lemma" of Wild et al (2015)).

The distribution of $s@\mathcal{G}$, conditional on $\mathbf{I}$, is given by the following urn model: We put all $n-k$ ordinary elements in an urn and draw their positions in $\mathbf{A}$. I_1 of the elements are colored red (namely the small ones), the rest is black (non-small). Now we draw the $\lvert\mathcal{G}\rvert = I_3$ elements in g's range from the urn without replacement. Then $s@\mathcal{G}$ is exactly the number of red (small) elements drawn and thus $s@\mathcal{G} \overset{\mathcal{D}}{=} \mathrm{HypG}(I_3, I_1, n-k)$.

The arguments for $l@\mathcal{K}$ are similar, however the additional δ in $|\mathcal{K}|$ needs special care. As shown in the proof of Lemma 3.7 of Wild et al (2015), the additional element in k's range for the case $\delta = 1$ is U_χ, which then is large by definition of δ. It thus simply contributes as additional summand: $l@\mathcal{K} \overset{\mathcal{D}}{=} \mathrm{HypG}(I_1 + I_2, I_3, n-k) + \delta$. Finally, the distribution of δ is Bernoulli $\mathrm{B}(\frac{I_3}{n-k})$, since conditional on $\mathbf{I}$, the probability of an ordinary element to be large is $I_3/(n-k)$.

5.2.1 Comparisons

Recall that we consider for C_n only the comparisons from the Partition procedure; as the sample size and the Insertionsort threshold are both constant, the number of other comparisons is bounded by $O(n)$ and can thus be ignored for the leading term of costs. It remains to count the comparisons during the first partitioning step, which we will denote by $T_C = T_C(n)$ instead of the generic toll T_n. Similarly, we will write T_S, T_{BC} and T_{SE} for the number of swaps, executed Bytecode instructions and scanned elements incurred in the first call to Partition.

One can approximate $T_C(n)$ on an abstract and intuitive level as follows: We need one comparison per ordinary element for sure, but some elements require a a second one to classify them as small, medium or large. Which elements are expensive and which are cheap (w. r. t. comparisons) depends on the index — either k or g — by which an element is reached: k first compares with P, so small elements are classified with only one comparison. Elements scanned by g are first compared with Q, so here the large ones are beneficial. Note that medium elements always need both comparisons. Using the notation introduced in Section 3.2, this gives a total of $(n-k) + I_2 + (l@\mathcal{K}) + (s@\mathcal{G})$ comparisons in the first partitioning step.

Some details of the partitioning algorithm are, however, easily overlooked at this abstract level of reasoning: a summand $+2\delta$ is missing in the above result. Essentially, the reason is that how much k overshoots g at the end of partitioning depends on the class of the element U_χ on which they meet. For the precise analysis, we therefore keep the argumentation closer to the actual algorithm at hand: for each location in the code where a key comparison is done, determine how often it is reached, then sum over all locations. The result is given in the following lemma.

Lemma 5.1: *Conditional on the partition sizes* $\mathbf{I}$*, the number of comparisons* $T_C = T_C(n)$ *in the first partitioning step of* $\mathrm{YQS}^w_{\mathbf{t}}$ *on a random permutation of size* $n > w$ *fulfills*

$$\begin{aligned} T_C(n) &= |\mathcal{K}| + |\mathcal{G}| + I_2 + (l@\mathcal{K}) + (s@\mathcal{G}) + \delta \\ &\overset{\mathcal{D}}{=} (n-k) + I_2 + \mathrm{HypG}(I_1 + I_2, I_3, n-k) \\ &\qquad + \mathrm{HypG}(I_3, I_1, n-k) + 3\mathrm{B}(\tfrac{I_3}{n-k}) \ . \end{aligned}$$

Proof: Each element that is accessed as $\mathbf{A}[k]$ or $\mathbf{A}[g]$ is directly compared (lines 3 and 8 of Algorithm 1), so we get $|\mathcal{K}| + |\mathcal{G}|$ "first" comparisons. The remaining contributions come from lines 7 and 11.

Line 7 is reached for every *non-small* element in k's range, giving a contribution of $(m@\mathcal{K}) + (l@\mathcal{K})$, where $m@\mathcal{K}$ denotes the number of medium elements in k's

range. Likewise, line 11 is executed for every non-large element in g's range, giving $(s@\mathcal{G})+(m@\mathcal{G})$ additional comparisons — but line 11 is also reached when the inner loop is left because of the second part of the loop condition, i.e., when the current element $\mathbf{A}[g]$ is large, but $k \geq g$. This can happen at most once since k and g have met then. It turns out that we get an additional execution of line 11 *if and only if* the element U_χ where k and g meet is large; this amounts to δ additional comparisons.

We never reach a medium element by both k and g because the only element that is potentially accessed through both indices is U_χ and it is only accessed via k in case $U_\chi > Q$, i.e., when it is *not* medium. Therefore, $(m@\mathcal{K}) + (m@\mathcal{G}) = I_2$, which proves the first equation. Wild et al (2015) give a more detailed explanation of the above arguments. The equality in distribution directly follows from Table 1. □

5.2.2 Swaps

As for comparisons, we only count the swaps in the partitioning step.

Lemma 5.2: *Conditional on the partition sizes* $\mathbf{I}$*, the number of swaps* $T_S = T_S(n)$ *in the first partitioning step of* $\mathrm{YQS}^w_{\mathbf{t}}$ *on a random permutation of size* $n > w$ *fulfills*

$$T_S(n) \;=\; I_1 \;+\; (l@\mathcal{K}) \;\overset{\mathcal{D}}{=}\; I_1 \;+\; \mathrm{HypG}(I_1 + I_2, I_3, n-k) \;+\; \mathrm{B}\left(\tfrac{I_3}{n-k}\right) .$$

Proof: No matter where a small element is located initially, it will eventually incur one swap that puts it at its final place (for this partitioning step) to the left of ℓ, see (Y3); this gives a contribution of I_1 swaps. The remaining swaps come from the "crossing pointer" scheme, where k stops on every large element on its way and g stops on all non-large elements. Whenever both k and g have stopped, the two out-of-order elements are exchanged in one swap (Y4). The number of such pairs is $l@\mathcal{K}$, which proves the first equation. The second equality follows from Table 1. □

5.2.3 Bytecode Instructions

A closer investigation of the partitioning method reveals the number of executions for every single Bytecode instruction in the algorithm. Details are omitted here; the analysis is very similar to the case without pivot sampling that is presented in detail in (Wild et al 2015).

Lemma 5.3: *Conditional on the partition sizes* $\mathbf{I}$*, the number of executed Java Bytecode instructions* $T_{BC} = T_{BC}(n)$ *of the first partitioning step of* $\mathrm{YQS}^w_{\mathbf{t}}$ *— implemented as in Appendix C of* (*Wild et al* 2015) *— fulfills on a random permutation of size* $n > w$

$$T_{BC}(n) \;\overset{\mathcal{D}}{=}\; 10n + 13I_1 + 5I_2 + 11\,\mathrm{HypG}(I_1 + I_2, I_3, n-k) + \mathrm{HypG}(I_1, I_1 + I_2, n-k) \;+\; O(1) . \quad \square$$

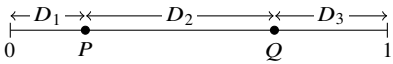


Figure 5 Graphical representation of the relation between $\mathbf{D}$ and the pivot values P and Q on the unit interval.

5.2.4 Scanned Elements

Lemma 5.4: *Conditional on the partition sizes* $\mathbf{I}$*, the number of scanned elements* $T_{SE} = T_{SE}(n)$ *in the first partitioning step of* $\mathrm{YQS}^{w}_{\mathbf{t}}$ *on a random permutation of size* $n > w$ *fulfills*

$$T_{SE}(n) \;=\; |\mathcal{K}| + |\mathcal{G}| + |\mathcal{L}| \;\overset{\mathcal{D}}{=}\; (n-k) \;+\; I_1 \;+\; \mathrm{B}\big(\tfrac{I_3}{n-k}\big)\ .$$

Proof: The first equality follows directly from the definitions: Our position sets include exactly the indices of array accesses. The equation in distribution is found using Table 1. □

5.2.5 Distribution of Partition Sizes

By (5) and the relation $\mathbf{J} = \mathbf{I} + \mathbf{t}$ between $\mathbf{I}$, the number of small, medium and large ordinary elements, and $\mathbf{J}$, the size of subproblems, we have $\mathbb{P}(\mathbf{I} = \mathbf{i}) = \binom{i_1+t_1}{t_1}\binom{i_2+t_2}{t_2}\binom{i_3+t_3}{t_3}\Big/\binom{n}{k}$. Albeit valid, this form results in nasty sums with three binomials when we try to compute expectations involving $\mathbf{I}$.

An alternative characterization of the distribution of $\mathbf{I}$ that is better suited for our needs exploits that we have i. i. d. $\mathcal{U}(0,1)$ variables. If we condition on the pivot *values*, i. e., consider P and Q fixed, an ordinary element U is small, if $U \in (0, P)$, medium if $U \in (P, Q)$ and large if $U \in (Q, 1)$. The lengths $\mathbf{D} = (D_1, D_2, D_3)$ of these three intervals (see Figure 5), thus are the *probabilities* for an element to be small, medium or large, respectively. Note that this holds *independently* of all other ordinary elements! The partition sizes $\mathbf{I}$ are then obtained as the collective outcome of $n - k$ independent drawings from this distribution, so conditional on $\mathbf{D}$, $\mathbf{I}$ is multinomially $\mathrm{Mult}(n-k, \mathbf{D})$ distributed.

With this alternative characterization, we have *decoupled* the pivot *ranks* (determined by $\mathbf{I}$) from the pivot *values*, which allows for a more elegant computation of expected values (see Appendix C). This decoupling trick has (implicitly) been applied to the analysis of classic Quicksort earlier, e. g., by Neininger (2001).

5.2.6 Distribution of Pivot Values

The input array is initially filled with n i. i. d. $\mathcal{U}(0,1)$ random variables from which we choose a sample $\{V_1, \dots, V_k\} \subset \{U_1, \dots, U_n\}$ of size k. The pivot values are then selected as order statistics of the sample: $P \coloneqq V_{(t_1+1)}$ and $Q \coloneqq V_{(t_1+t_2+2)}$ (cf. Section 3.1). In other words, $\mathbf{D}$ is the vector of *spacings* induced by the order statistics $V_{(t_1+1)}$ and $V_{(t_1+t_2+2)}$ of k i. i. d. $\mathcal{U}(0,1)$ variables $V_1, \dots, V_k$, which is known to have a *Dirichlet* $\mathrm{Dir}(\mathbf{t}+1)$ distribution (Proposition B.1).

6 Average-Case Analysis

6.1 Expected Partitioning Costs

In Section 5, we characterized the full distribution of the costs of the first partitioning step. However, since those distributions are *conditional* on other random variables, we have to apply the *law of total expectation*. By linearity of the expectation, it suffices to consider the summands given in the following lemma.

Lemma 6.1: *For pivot sampling parameter* $\mathbf{t} \in \mathbb{N}^3$ *and partition sizes* $\mathbf{I} \stackrel{\mathcal{D}}{=} \mathrm{Mult}(n-k, \mathbf{D})$*, based on random spacings* $\mathbf{D} \stackrel{\mathcal{D}}{=} \mathrm{Dir}(\mathbf{t}+1)$*, the following* (*unconditional*) *expectations hold*:

$$\begin{aligned}
\mathbb{E}[I_j] &= \frac{t_j+1}{k+1}(n-k), && (j=1,2,3),\\
\mathbb{E}[\mathrm{B}(\tfrac{I_3}{n-k})] &= \frac{t_3+1}{k+1} = \Theta(1), && (n\to\infty),\\
\mathbb{E}[\mathrm{HypG}(I_3, I_1, n-k)] &= \frac{(t_1+1)(t_3+1)}{(k+1)(k+2)}(n-k-1),\\
\mathbb{E}[\mathrm{HypG}(I_1+I_2, I_3, n-k)] &= \frac{(t_1+t_2+2)(t_3+1)}{(k+1)(k+2)}(n-k-1)\,.
\end{aligned}$$

Using known properties of the involved distributions, the proof is an elementary computation. It is given in detail in Appendix C for interested readers.

The direct consequence of Lemma 6.1 is that for all our cost measures, we have expected partitioning costs of the form $\mathbb{E}[T_n] = an + b$ with constants a and b.

6.2 Solution of the Recurrence

By taking expectations on both sides of the distributional recurrence (Equation (4) on page 15), we obtain an ordinary recurrence for the sequence of expected costs $(\mathbb{E}[C_n])_{n\in\mathbb{N}}$. We solve this recurrence using Roura's *Continuous Master Theorem* (*CMT*) (Roura 2001), but first give an informal derivation of the solution to convey the main intuition behind the CMT. Precise formal arguments are then given in Appendix D.

6.2.1 Rewriting the Recurrence

To solve the recurrence, it is convenient first to rewrite Equation (4) a little. We start by *conditioning* on $\mathbf{J}$. For $n > w$, this gives

$$C_n \stackrel{\mathcal{D}}{=} T_n + \sum_{j=0}^{n-2}\Big(\mathbb{1}_{\{J_1=j\}}C_j + \mathbb{1}_{\{J_2=j\}}C'_j + \mathbb{1}_{\{J_3=j\}}C''_j\Big)\,.$$

Taking expectations on both sides and exploiting independence yields

$$\mathbb{E}[C_n] \;=\; \begin{cases} \mathbb{E}[T_n] \;+\; \displaystyle\sum_{j=0}^{n-2} \mathbb{E}[C_j] \sum_{r=1}^{3} \mathbb{P}(J_r = j) & \text{for } n > w; \\ \mathbb{E}[W_n], & \text{for } n \le w. \end{cases} \tag{6}$$

By definition, $J_r = I_r + t_r$ and, conditional on $\mathbf{D}$, I_r is $\mathrm{Bin}(n-k, D_r)$ distributed for $r = 1, 2, 3$. (The marginal distribution of a multinomial vector is the binomial distribution.) We thus have conditional on $\mathbf{D}$ that

$$\mathbb{P}(I_r = i) \;=\; \binom{n-k}{i} D_r^i (1 - D_r)^{n-k-i}$$

and upon unconditioning

$$\mathbb{P}(J_r = j) \;=\; \binom{n-k}{j-t_r} \mathbb{E}_{\mathbf{D}}\left[D_r^{j-t_r} (1 - D_r)^{n-k-j+t_r} \right].$$

There are three cases to distinguish depending on the toll function, which are well-known from the classical *master theorem* for divide-and-conquer recurrences:

1. If the toll function grows very fast with n, the first recursive call will dominate overall costs, as the toll costs of subproblems are small in relation to the first step.
2. On the other hand, if the toll function grows very slow with n, the topmost calls will be so cheap in relation that the *number* of base case calls on constant size subproblems will dictate overall costs.
3. Finally, for toll functions of just the right rate of growth, the recursive calls on each level of the recursion tree sum up to (roughly) the same cost and the overall solution is given by this sum of costs times the recursion depth.

Binary search and Mergesort are prime examples of the third case, in the analysis of Karatsuba's integer multiplication or Strassen's matrix multiplication, we end up in the second case and in the Median-of-Medians selection algorithm the initial call is asymptotically dominating and we get the first case (see, e.g., Cormen et al (2009)).

Our Equation (6) shows essentially the same three cases depending on the asymptotic growth of $\mathbb{E}[T_n]$. The classical master theorem distinguishes the cases by comparing, for large n, the toll of the topmost call with the total tolls of all its immediate child recursive calls. If there is an (asymptotic) imbalance to the one or the other side, this imbalance will eventually dominate for large n. The same reasoning applies to our recurrence, only that computations become a little trickier since the subproblem sizes are not fixed a priori.

Let us first symbolically substitute zn for j in (6), so that $z \in [0, 1]$ becomes the *relative subproblem size*:

$$\mathbb{E}[C_n] \;=\; \mathbb{E}[T_n] \;+\; \sum_{zn=0}^{n-2} \mathbb{E}[C_{zn}] \sum_{r=1}^{3} \mathbb{P}\left(\frac{J_r}{n} = z \right).$$

In the sum over zn, n of course remains unchanged, and z moves 0 towards 1. When n gets larger and larger, z "scans" the unit interval more and more densely, so that it is plausible to approximate the sum by an integral:

$$\sum_{zn=0}^{n-2} \mathbb{E}[C_{zn}] \sum_{r=1}^{3} \mathbb{P}\Big(\frac{J_r}{n} = z\Big) \;\approx\; \int_{z=0}^{1} \mathbb{E}[C_{zn}] \sum_{r=1}^{3} \mathbb{P}\Big(\frac{J_r}{n} = z \pm \frac{1}{2n}\Big)\, dz\,.$$

This idea has already been used by van Emden (1970) to compute the number of comparisons for classic Quicksort with median-of-three — in fact he was the first to derive that number analytically. However, some continuity assumptions are silently made in this step and a rigorous derivation has to work out the error terms that we make by this approximation. We defer a formal treatment of these issues to Appendix D.

Finally, $J_r = I_r + t_r$ has the expectation $\mathbb{E}[J_l \mid \mathbf{D}] = D_l\, n + t_r$ conditional on $\mathbf{D}$ and so for large n

$$\sum_{zn=0}^{n-2} \mathbb{E}[C_{zn}] \sum_{r=1}^{3} \mathbb{P}\Big(\frac{J_r}{n} = z\Big) \;\approx\; \int_{z=0}^{1} \mathbb{E}[C_{zn}] \sum_{r=1}^{3} \mathbb{P}\Big(D_r = z \pm \tfrac{1}{2n}\Big)\, dz\,.$$

Intuitively, this means that the *relative subproblem sizes* in dual-pivot Quicksort with pivot sampling parameter $\mathbf{t}$ have a Dirichlet distribution with parameters $\mathrm{Dir}(t_1 + 1, k - t_1)$, $\mathrm{Dir}(t_2 + 1, k - t_2)$ and $\mathrm{Dir}(t_3 + 1, k - t_3)$, respectively. The main advantage of this last form is that the integral does not depend on n anymore and we obtain the following *continuous* recurrence for $\mathbb{E}[C_n]$:

$$\mathbb{E}[C_n] \;\approx\; \mathbb{E}[T_n] \;+\; \int_0^1 w(z)\, \mathbb{E}[C_{zn}]\, dz\,, \tag{7}$$

for a "shape function" $w(z) \coloneqq \sum_{r=1}^{3} f_{D_r}(z)$ where f_{D_r} is the density function of the $\mathrm{Dir}(t_r + 1, k - t_r)$ distribution.

6.2.2 Which Case of the Master Theorem?

We are now in the position to compare the toll of the first call $\mathbb{E}[T_n]$ to the total tolls of its child recursive calls, i.e., how

$$\int_0^1 \mathbb{E}[T_{zn}]\, w(z)\, dz \tag{8}$$

relates to $\mathbb{E}[T_n]$. We assume $\mathbb{E}[T_n] = an + O(n^{1-\epsilon})$ for $\epsilon > 0$, which for our cost measures is fulfilled with $\epsilon = 1$. As $\mathbb{E}[C_n]$ is linear in $\mathbb{E}[T_n]$, we can solve the recurrence for the leading term an and the error term $O(n^{1-\epsilon})$ separately. When working out the integrals, it turns out that

$$\int_0^1 a\, zn\, w(z)\, dz \;=\; an\,, \tag{9}$$

so the last case from above applies: The total cost of the child subproblems is (asymptotically) the same as the cost of the initial call. In analogy with the classical master theorem, the overall costs $\mathbb{E}[C_n]$ are thus the toll cost of the initial call times the number of levels in the recursion tree.

6.2.3 Solve by Ansatz

Guessing that the number of recursion levels will be logarithmic as in the case of the classical master theorem, we make the *ansatz* $\mathbb{E}[C_n] = \frac{a}{\eta} n \ln n$ with an unknown constant η. Inserting into the continuous recurrence (7) yields

$$\frac{a}{\eta} n \ln n \quad = \quad an + \int_0^1 w(z) \, \frac{a}{\eta} zn \ln(zn) \, dz \; .$$

Multiplying by $\frac{\eta}{an}$ and rearranging, we find

$$\eta \quad = \quad \ln n \cdot \left(1 - \textstyle\int_0^1 zw(z) \, dz\right) \; - \; \displaystyle\int_0^1 z \ln(z) w(z) \, dz \, ,$$

where the first integral is 1 (see (9)), which is good since otherwise the "constant" η would involve $\ln n$. The second integral turns out to be precisely $-\mathcal{H}$, for $\mathcal{H} = \mathcal{H}(\mathbf{t})$ the discrete entropy of $\mathbf{t}$ defined in Equation (1) and so

$$\mathbb{E}[C_n] \quad = \quad \frac{a}{\mathcal{H}} n \ln n$$

fulfills the continuous recurrence (7) exactly.

Working out the error terms that we get by approximating the sum of the original recurrence by an integral and by approximating the weights in the discrete recurrence by the shape function $w(z)$, we obtain the following theorem.

Theorem 6.2: *Let* $\mathbb{E}[C_n]$ *be a sequence of numbers satisfying Equation* (6) *on page* 21 *for* $\mathbf{t} \in \mathbb{N}^3$ *and a constant* $w \geq k = t_1 + t_2 + t_3 + 2$ *and let the toll function* $\mathbb{E}[T_n]$ *be of the form* $\mathbb{E}[T_n] = an + O(n^{1-\epsilon})$ *for constants* a *and* $\epsilon > 0$. *Then we have* $\mathbb{E}[C_n] \sim \frac{a}{\mathcal{H}} \, n \ln n$, *where* $\mathcal{H}$ *is given by Equation* (1) *on page* 13.

A slightly weaker form of Theorem 6.2 has first been proven by Hennequin (1991, Proposition III.9) using direct arguments on the Cauchy-Euler differential equations that the recurrence implies for the generating function of $\mathbb{E}[C_n]$. Building on the toolbox of handy and ready-to-apply theorems developed by the analysis-of-algorithms community, we can give a rather concise and elementary proof making our informal derivation from above precise: Appendix D gives the detailed argument for solving the recurrence using the *Continuous Master Theorem* by Roura (2001). An alternative tool that remains closer to Hennequin's original arguments is offered by Chern et al (2002).

Theorem 4.1 now directly follows by using Lemma 6.1 on the partitioning costs from Lemma 5.1, 5.2 and 5.3 and plugging the result into Theorem 6.2.

7 Validation

The purpose of this paper is to approach an explanation for the efficiency of Yaroslavskiy's Quicksort in practice using the methods of the mathematical analysis of algorithms, which means that we define a *model* of the actual program (given by our

Algorithm 2) and its *costs*. For the latter, different cost measures have proven valuable for different purposes, so we consider several of them. As in the natural sciences, our model typically loses some details of the "real world", which means that we make a *modeling error*. For example, counting scanned elements comes close to, but is not the same as counting actual cache misses, see Section 7.2.

On top of that, the precise analysis of the model of an algorithm can still be infeasible or at least overly complicated. For example in our recurrence (6), rather elementary means sufficed to determine the leading term of an asymptotic expansion of the solution; obtaining more terms of the expansion is much harder, though. Luckily, one can often resort to such asymptotic approximations for $n \to \infty$ without losing too much accuracy for practical input sizes; yet we do make an *analysis error* whenever we use asymptotics, see Section 7.1.

To assess the predictive quality of our analysis, we compare our results to some practical values. Wherever possible, we try to separate modeling errors from analysis errors to indicate whether further effort should be put in a more detailed analysis of the present model or in a refined model.

As discussed in Section 3.3, Algorithm 2 should be considered an "academic" program, which is tailor-made for analysis, not for productive use and therefore, we do not report running times. Other works contain actual running times of (more) realistic implementations: Wild (2012) investigates the basic variants without pivot sampling. Wild et al (2013) compare different choices for the pivots from a sample of size $k = 5$. Aumüller and Dietzfelbinger (2013) compare several variants with and without pivot sampling and also other dual-pivot partitioning methods. Moreover, Kushagra et al (2014) include a three-pivot Quicksort and report measured cache misses as well (see also Section 7.2).

7.1 Quality of Asymptotic Approximations

In this section, we focus on the analysis error. To obtain values to compare the asymptotic approximations with, we implemented $\mathrm{YQS}_{\mathbf{t}}^{w}$ (as given in Algorithm 2) and augmented the code to count key comparisons, swaps and scanned elements. For counting the number of executed Java Bytecode instructions, we used our tool *MaLiJAn*, which can automatically generate code to count the number of Bytecodes (Wild et al 2013).

All reported counts are averages of runs on 1000 random permutations of the same size. We use powers of 2 as input sizes and the plots show n on a logarithmic x-axis. The y-axis is normalized by dividing by $n \ln n$.

For an actual execution, one has to fix the parameters $\mathbf{t}$ and w. We experimented with several choices, but found the quality of the asymptotic expansions to be very stable w. r. t. moderate values of $\mathbf{t}$, i. e., for sample sizes up to $k = 11$. Unless otherwise stated, all plots below show the *tertiles-of-five* choice $\mathbf{t} = (1, 1, 1)$. For the Insertionsort threshold w, values used in practice ($w = 46$ for Oracle's Java 7 library) yield a significant influence on overall costs for moderate n, see Figure 6. This contribution is completely ignored in the leading term, and thus the predictive quality of the

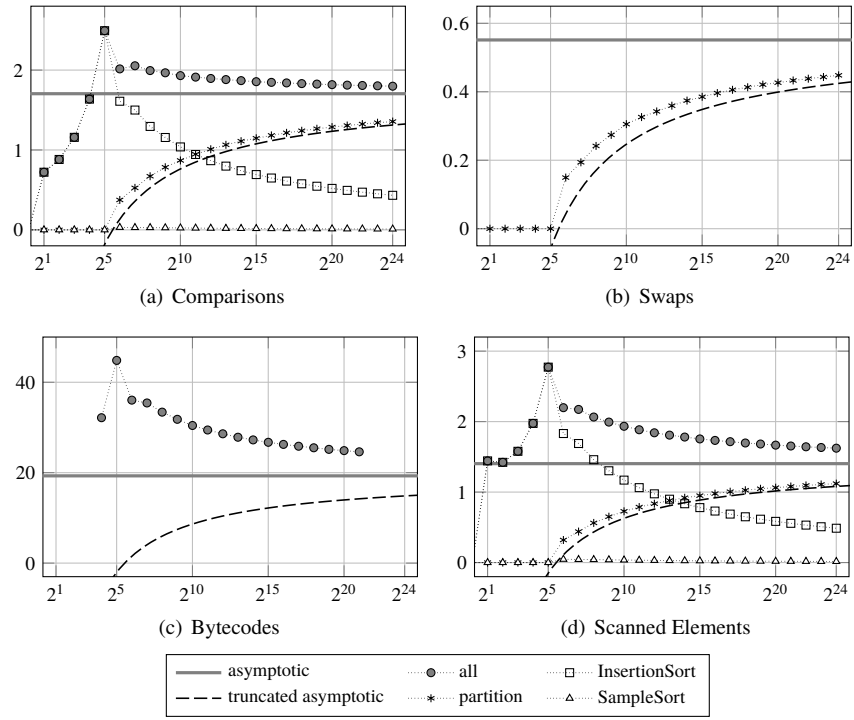


Figure 6 Comparison, swap, Bytecode and scanned element counts (·●·) normalized by $n \ln n$, for $\mathrm{YQS}^w_{\mathbf{t}}$ with $\mathbf{t} = (1, 1, 1)$ and $w = 46$ against the leading-term asymptotic $\frac{a}{\mathcal{H}} n \ln(n)$ (——) from Theorem 4.1 and its truncated version $\frac{a}{\mathcal{H}} n \ln(\frac{n}{w})$ (– – –). For comparisons and scanned elements, the contributions from Partition (·∗·), InsertionSort (·□·) and SampleSort (·△·) are also given separately. Note that swaps only occur during partitioning (Insertionsort uses single write accesses). For reasonably large n, the main contribution indeed comes from Partition, however, InsertionSort on short subarrays also contributes significantly. This is probably true for all cost measures, even though not shown here in detail.

asymptotic is limited for large values of w. For $w = 7$, the analysis error is much smaller, but still clearly visible, see Figure 7.

In plain numbers, we have with $w = 46$ and input size $n = 2^{20} \approx 10^6$ around 5% error for comparisons, 28% error in the number of swaps, 23% for Bytecodes and 16% error for scanned elements. For $w = 7$, the errors are 9%, 6%, 15% and 1% for comparisons, swaps, Bytecodes and scanned elements, respectively.

Although a complete derivation of the linear term of costs is out of the question here, a simple heuristic allows to improve the predictive quality of our asymptotic formulas for the partitioning costs. The main error that we make is to ignore that Partition is not called at all for subarrays of size at most w. We can partially correct for that by truncating the recursion tree at level $\ln(\frac{n}{w})$, instead of going down all $\ln(n)$ levels, i. e., instead of total costs $\frac{a}{\mathcal{H}} n \ln n$, we use the *truncated term* $\frac{a}{\mathcal{H}} n \ln(\frac{n}{w})$. (This means that the last $\ln(w)$ levels of the recursion tree are subtracted from the leading term.) The plots in this section always include the pure leading term as a straight

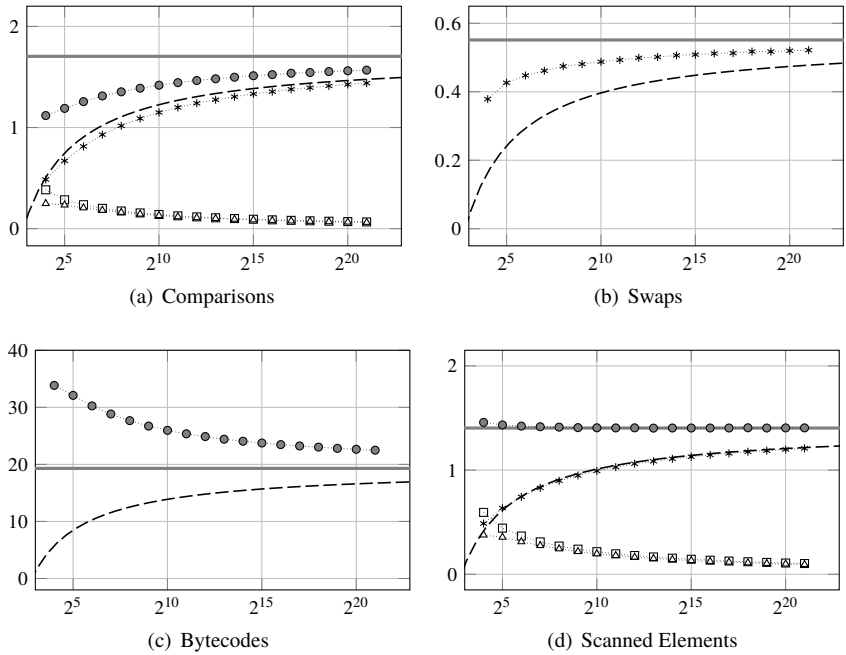


Figure 7 Same as Figure 6, but with smaller Insertionsort threshold $w = 7$.

black line and the truncated term as a dashed black line. It is clearly visible that the truncated term gives a much better approximation of the costs from Partition.

Of course, the above argument is informal reasoning on an oversimplified view of the recurrence; the actual recursion tree does neither have exactly $\ln(n)$ levels, nor are all levels completely filled. Therefore, the truncated term does not give the correct linear term for partitioning costs, and it completely ignores the costs of sorting the short subarrays by Insertionsort. It is thus to be expected that the truncated term is *smaller* than the actual costs, whereas the leading term alone often lies above them.

7.2 Scanned Elements vs. Cache Misses

This section considers the modeling error present in our cost measures. Comparisons, swaps and Bytecodes are precise by definition; they stand for themselves and do not model more intricate practical costs. (They were initially intended as models for running time, but as discussed in the introduction were already shown to fail in explaining observed running time differences.) The number of scanned elements was introduced in this paper as a model for the number of cache misses in Quicksort, so we ought to investigate the difference between the two.

The problem with cache misses is that in practice there are multiple levels of caches and that cache sizes, block sizes, eviction strategies and associativity all differ from machine to machine. Moreover, block borders in a hardware cache are aligned

with physical address blocks (such that one can use the first few bits as cache block address), so the precise caching behavior depends on the starting address of the array that we are sorting; not to speak of the influence other processes have on the content of the cache ...

We claim, though, that such details do not have a big impact on the overall number of cache misses in Quicksort and focus in this paper on an *idealized cache*, i.e., a fully associative cache (i.e., no block address aliasing) that uses the *least-recently-used* (*LRU*) eviction strategy. The cache synchronizes itself with main memory in blocks of B consecutive array elements and it can hold up to M array entries in total, where $M \geq B$ is a multiple of B. Moreover, we assume that our array always starts at a block boundary, that its length is a multiple of the block size and that the cache is initially empty. We then simulated Quicksort on such an idealized cache, precisely counting the number of incurred cache misses, i.e., of accesses to indices of the array, whose block is currently not in the cache.

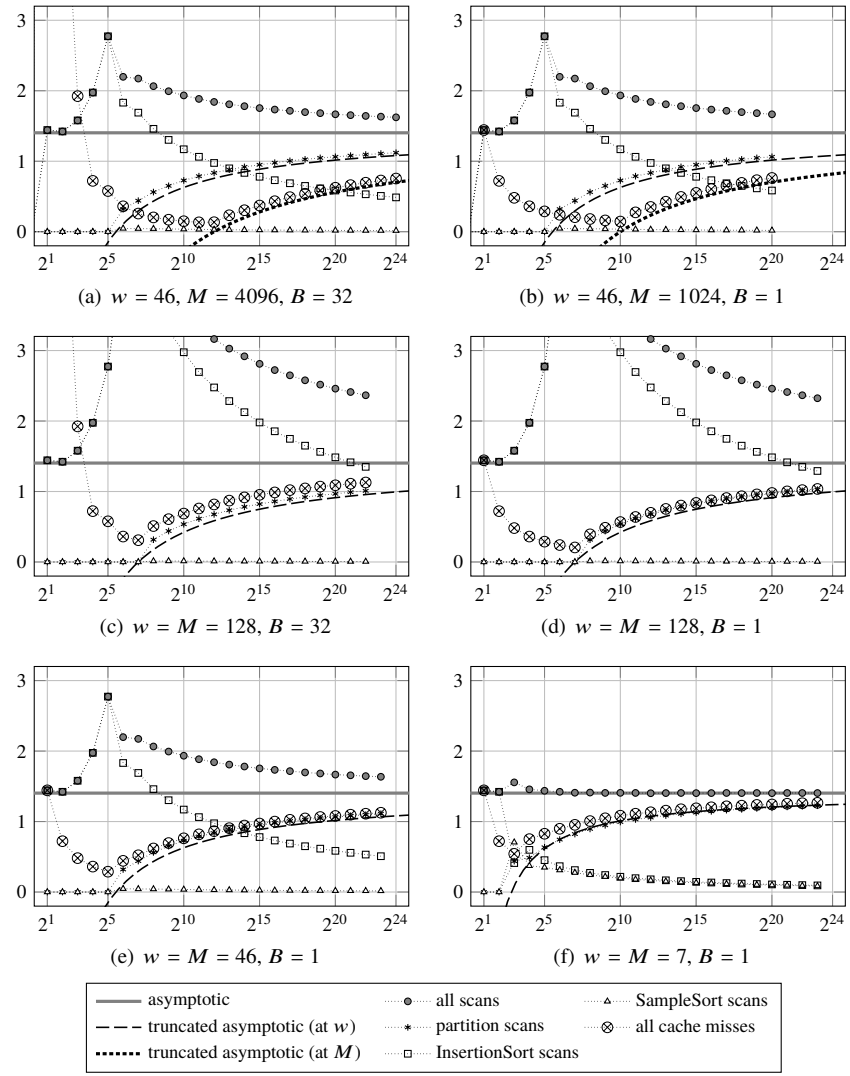
The resulting cache miss counts (averages of 1000 runs) are shown in Figure 8 for a variety of parameter choices. At first sight, the overall picture seems rather disappointing: the total number of scanned elements and the number of cache misses do not seem to match particularly well (filled circles and crossed circles in Figure 8). The reason is that once the subproblem size is at most M, the whole subarray fits into the cache and at most M/B additional cache misses suffice for sorting the whole subarray; whereas in terms of scanned elements, the contribution of these subarrays is at least linearithmic[3] (for partitioning) or even quadratic (for Insertionsort).

If, however, the cache size M and the Insertionsort threshold w are the same (as in Figure 8(c) – (f)), the number of cache misses and the number of scanned elements agree very well, if we count the latter in procedure Partition only. If we consider the asymptotic for the number of scanned elements, but truncate the recursion to $\ln(\frac{n}{M})$ levels (cf. Section 7.1), we find a very good fit to the number of cache misses (see dotted lines resp. dashed lines in Figure 8). From that we can conclude that (a) the main error made in counting scanned elements is to ignore the cutoff at M and that (b) the base cases (subproblems of size at most M) have little influence and can be ignored for performance prediction. We also note that $\frac{a_{SE}}{\mathcal{H}} \frac{n}{B} \ln(\frac{n}{M})$ is a very good approximation for the overall number of cache misses for *all* our parameter choices for M, B and w (even if the number of blocks M/B that fit in the cache at the same time is as small as 4, see Figure 8(c)).

The most important algorithmic conclusion from these findings is that *we can safely use the number of scanned elements to compare different Quicksort variants*; the major part of the modeling error, that we make in doing so, will cancel out when comparing two algorithms.

Kushagra et al (2014) immediately report the truncated term as an asymptotic upper bound for the number of cache misses. We think that it is worthwhile to have the clean separation between the mathematically precise analysis of scanned elements and the machine-dependent cache misses in practice — we can now compare Quicksort variants in terms of scanned elements instead of actual cache misses, which is a much more convenient cost measure to deal with.

[3] We use the neologism "linearithmic" to say that a function has order of growth $\Theta(n \log n)$.



(a) $w = 46$, $M = 4096$, $B = 32$ (b) $w = 46$, $M = 1024$, $B = 1$

(c) $w = M = 128$, $B = 32$ (d) $w = M = 128$, $B = 1$

(e) $w = M = 46$, $B = 1$ (f) $w = M = 7$, $B = 1$

Figure 8 Comparison of cache miss counts (⊗) from our idealized fully-associative LRU cache with different cache and block sizes M resp. B with corresponding scanned element counts (●). The latter are also given separately for Partition (∗), InsertionSort (□) and SampleSort (△). To make the counts comparable, the number of cache misses has been multiplied by B. All plots are normalized by $n \ln n$ and show results for $\mathrm{YQS}_{\mathbf{t}}^{w}$ with $\mathbf{t} = (1, 1, 1)$ and different Insertionsort thresholds w. The fat line (——) shows the leading-term asymptotic for scanned elements from Theorem 4.1, namely $\frac{80}{57} n \ln n$. The dashed line (– – –) is the truncated term $\frac{80}{57} n \ln(\frac{n}{w})$ and the dotted line (·····) shows $\frac{80}{57} n \ln(\frac{n}{M})$, which is the leading term truncated at subproblems that fit into the cache.

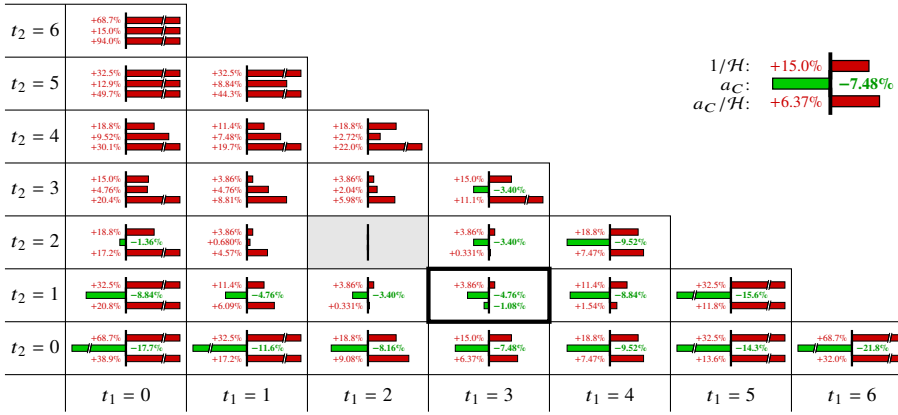


Figure 9 Inverse of discrete entropy (top), number of comparisons per partitioning step (middle) and overall comparisons (bottom) for all $\mathbf{t}$ with $k = 8$, relative to the tertiles case $\mathbf{t} = (2, 2, 2)$.

$t_1 \backslash t_2$	0	1	2	3
0	1.9956	1.8681	2.0055	2.4864
1	1.7582	**1.7043**	1.9231	
2	1.7308	1.7582		
3	1.8975			

(a) $a_C/\mathcal{H}$

$t_1 \backslash t_2$	0	1	2	3
0	0.4907	0.4396	0.4121	**0.3926**
1	0.6319	0.5514	0.5220	
2	0.7967	0.7143		
3	1.0796			

(b) $a_S/\mathcal{H}$

$t_1 \backslash t_2$	0	1	2	3
0	20.840	**18.791**	19.478	23.293
1	20.440	19.298	21.264	
2	22.830	22.967		
3	29.378			

(c) $a_{BC}/\mathcal{H}$

$t_1 \backslash t_2$	0	1	2	3
0	1.6031	**1.3462**	**1.3462**	1.6031
1	1.5385	1.4035	1.5385	
2	1.7308	1.7308		
3	2.2901			

(d) $a_{SE}/\mathcal{H}$

Table 2 $\frac{a_C}{\mathcal{H}}$, $\frac{a_S}{\mathcal{H}}$, $\frac{a_{BC}}{\mathcal{H}}$ and $\frac{a_{SE}}{\mathcal{H}}$ for all $\mathbf{t}$ with $k = 5$. Rows resp. columns give t_1 and t_2; t_3 is then $k - 2 - t_1 - t_2$. The symmetric choice $\mathbf{t} = (1, 1, 1)$ is shaded, the minimum is printed in bold.

8 Discussion

8.1 Asymmetries Everywhere

With Theorem 4.1, we can find the optimal sampling parameter $\mathbf{t}$ for any given sample size k. As an example, Figure 9 shows $\mathcal{H}$, a_C and the overall number of comparisons for all possible $\mathbf{t}$ with sample size $k = 8$: The discrete entropy decreases symmetrically as we move away from the center $\mathbf{t} = (2, 2, 2)$; this corresponds to the effect of less evenly distributed subproblem sizes. The individual partitioning steps, however, are cheap for *small* values of t_2 and optimal in the extreme point $\mathbf{t} = (6, 0, 0)$. For minimizing the *overall* number of comparisons — the ratio of latter — we have to find a suitable trade-off between the center and the extreme point $(6, 0, 0)$; in this case the minimal total number of comparisons is achieved with $\mathbf{t} = (3, 1, 2)$.

Apart from this trade-off between the evenness of subproblem sizes and the number of comparisons per partitioning, Table 2 shows that the optimal choices for $\mathbf{t}$ w. r. t. comparisons, swaps, Bytecodes and scanned elements heavily differ. The partitioning costs are, in fact, in *extreme conflict* with each other: for all $k \geq 2$, the minimal values of a_C, a_S and a_{BC} among all choices of $\mathbf{t}$ for sample size k are attained for $\mathbf{t} = (k-2, 0, 0)$, $\mathbf{t} = (0, k-2, 0)$, $\mathbf{t} = (0, 0, k-2)$ and $\mathbf{t} = (0, t, k-2-t)$ for $0 \leq t \leq k-2$, respectively. Intuitively this is because the strategy minimizing partitioning costs in isolation executes the cheapest path through the partitioning loop as often as possible, which naturally leads to extreme choices for $\mathbf{t}$. It then depends on the actual numbers, where the total costs are minimized. It is thus not possible to minimize all cost measures at once, and the rivaling effects described above make it hard to reason about optimal parameters merely on a qualitative level.

8.2 Optimal Order Statistics for fixed k

Given any cost measure we can compute — although not in closed form — the optimal sampling parameter $\mathbf{t}^*$ for a fixed size of the sample $k = k(\mathbf{t})$. Here, by optimal sampling parameter we mean the parameter $\mathbf{t}^* = (t_1^*, t_2^*, t_3^*)$ that minimizes the leading term of the corresponding cost, that is, the choice minimizing $q_X := a_X/\mathcal{H}$ (where X is C, S, BC, or SE). Table 3 lists the optimal sampling parameters of $\mathrm{YQS}_{\mathbf{t}}^{w}$ for several values of k of the form $k = 3\lambda + 2$ (as well as $k = 100$).

In Section 8.3 we explore how $\mathbf{t}^*$ evolves as $k \to \infty$: for each cost measure there exists an optimal parameter $\boldsymbol{\tau}^* = \lim_{k\to\infty} \mathbf{t}^*/k$. For finite k several remarks are in order; the most salient features of $\mathbf{t}^*$ can be easily spotted from a short table like Table 3.

First, for swaps the optimal sampling parameter is always $\mathbf{t}^* = (0, k-2, 0)$ ($(0, 0, k-2)$ is also optimal) and

$$q_S^* = \frac{2k(k+1)}{(2kH_k - 1)(k+2)}.$$

Indeed, as far as swaps are concerned, pivot P should be as small as possible while pivot Q is as large as possible, for then the expected number of swaps in a single partitioning step is $2/(k+2)$.

For comparisons it is not true that a balanced sampling parameter $\mathbf{t} = (\lambda, \lambda, \lambda)$ (when $k = 3\lambda + 2$) is the best choice, except for $\lambda = 1$. For instance, for $k = 8$ we have $\mathbf{t}^* = (3, 1, 2)$. The behavior of $\mathbf{t}^*$ as k increases is somewhat erratic, although it quickly converges to $\approx (0.43k, 0.27k, 0.3k)$ (cf. Section 8.3).

For Bytecodes and scanned elements, the optimal sampling parameters are even more biased. They are not very different from each other.

In the case of scanned elements, if $\mathbf{t} = (t_1, t_2, t_3)$ is optimal so is $\mathbf{t}' = (t_1, t_3, t_2)$ (since $\mathcal{H}$ is symmetric in t_1, t_2 and t_3 and a_{SE} is symmetric in t_2 and t_3). The optimal choice for scanned elements seems always to be of the form (t_1, t_2, t_2) or $(t_1, t_2, t_2 + 1)$ (or $(t_1, t_2 + 1, t_2)$).

Assuming that the optimal parameter is of the form $\mathbf{t}^* = (t_1, t_2, t_2)$ with $t_2 = (k - 2 - t_1)/2$ we can obtain an approximation for the optimal t_1^* by looking at

k	Cost measure	$\mathbf{t}^*$	$q_X = \frac{a_X}{\mathcal{H}}$
no sampling	comparisons	(0,0,0)	1.9
	swaps	(0,0,0)	0.6
	Bytecodes	(0,0,0)	21.7
	scanned elements	(0,0,0)	1.6
5	comparisons	(1,1,1)	1.70426
	swaps	(0,3,0)	0.392585
	Bytecodes	(0,1,2)	18.7912
	scanned elements	(0,1,2)	1.34615
8	comparisons	(3,1,2)	1.62274
	swaps	(0,6,0)	0.338937
	Bytecodes	(1,2,3)	17.8733
	scanned elements	(1,2,3)	1.27501
11	comparisons	(4,2,3)	1.58485
	swaps	(0,9,0)	0.310338
	Bytecodes	(2,3,4)	17.5552
	scanned elements	(1,4,4)	1.22751
17	comparisons	(6,4,5)	1.55535
	swaps	(0,15,0)	0.277809
	Bytecodes	(3,5,7)	17.1281
	scanned elements	(2,6,7)	1.19869
32	comparisons	(13,8,9)	1.52583
	swaps	(0,30,0)	0.240074
	Bytecodes	(6,10,14)	16.7888
	scanned elements	(5,12,13)	1.16883
62	comparisons	(26,16,18)	1.51016
	swaps	(0,60,0)	0.209249
	Bytecodes	(12,21,27)	16.5914
	scanned elements	(10,25,25)	1.15207
100	comparisons	(42,26,30)	1.50372
	swaps	(0,98,0)	0.19107
	Bytecodes	(20,34,44)	16.513
	scanned elements	(16,41,41)	1.14556

Table 3 Optimal sampling parameter $\mathbf{t}^*$ for the different cost measures and several fixed values of the sample size k.

$q_{SE} = a_{SE}/\mathcal{H}$ as a continuous function of its arguments and substituting H_n by $\ln(n)$: taking derivatives w. r. t. t_1, and solving $dq_{SE}/dt_1 = 0$ gives us $t_1^* \approx (3 - 2\sqrt{2})k$. Indeed, $\mathbf{t} = (t_1, t_2, k - 2 - t_1 - t_2)$ with

$$t_1 = \lfloor q^2(k-2) \rfloor, \qquad t_2 = \lceil q(k-2) \rceil \quad \text{and} \quad q = \sqrt{2} - 1$$

is the optimal sampling parameter for most k (in particular for all values of k in Table 3).

It is interesting to note in this context that the implementation in Oracle's Java 7 runtime library — which uses $\mathbf{t} = (1, 1, 1)$ — executes asymptotically *more* Bytecodes and needs more element scans (on random permutations) than $\mathrm{YQS}_{\mathbf{t}}^{w}$ with $\mathbf{t} = (0, 1, 2)$, despite using the same sample size $k = 5$. Whether this also results in a performance gain in practice, however, depends on details of the runtime environment (Wild et al 2013). (One should also note that the savings are only 2% respectively 4%.) Since these two cost measures, Bytecodes and scanned elements, are arguably the ones with highest impact on running time, it is very good news from the practitioner's point of view that the optimal choice for one of them is also reasonably good for the other; such choice should yield a close-to-optimal running time (as far as sampling is involved).

8.3 Continuous ranks

It is natural to ask for the optimal *relative ranks* of P and Q if we are not constrained by the discrete nature of pivot sampling. In fact, one might want to choose the sample size depending on those optimal relative ranks to find a discrete order statistic that falls close to the continuous optimum.

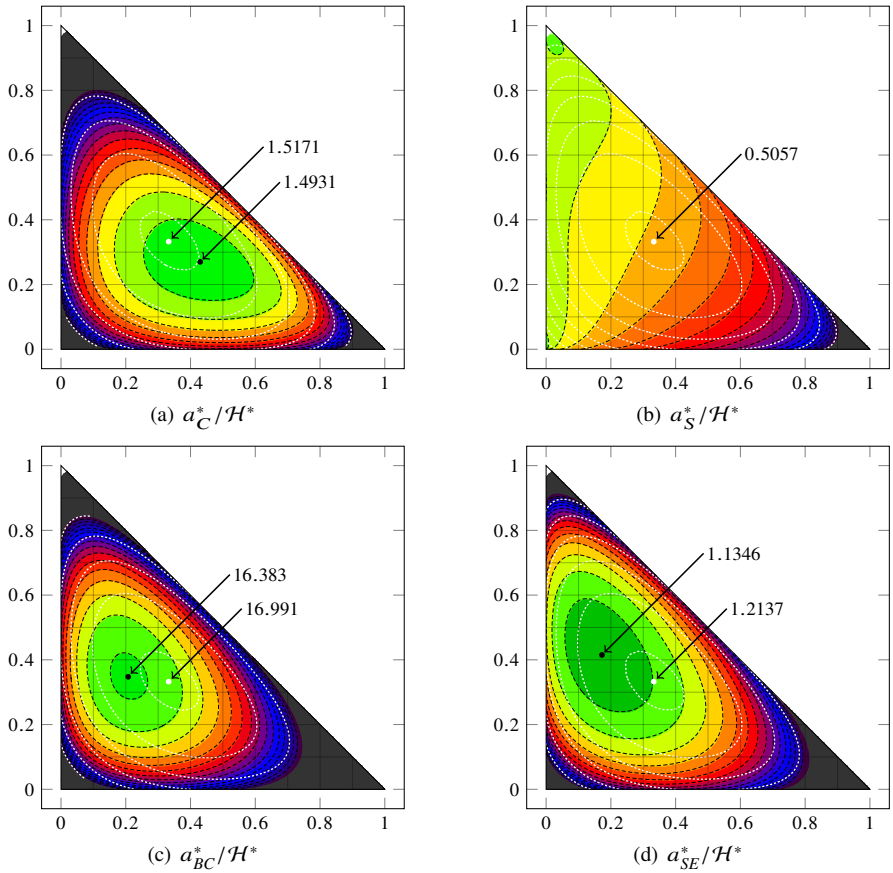
We can compute the optimal relative ranks by considering the limiting behavior of $\mathrm{YQS}_{\mathbf{t}}^{w}$ as $k \to \infty$. Formally, we consider the following family of algorithms: let $(t_r^{(j)})_{j \in \mathbb{N}}$ for $r = 1, 2, 3$ be three sequences of non-negative integers and set

$$k^{(j)} := t_1^{(j)} + t_2^{(j)} + t_3^{(j)} + 2$$

for every $j \in \mathbb{N}$. Assume that we have $k^{(j)} \to \infty$ and $t_r^{(j)}/k^{(j)} \to \tau_r$ with $\tau_l \in [0, 1]$ for $r = 1, 2, 3$ as $j \to \infty$. Note that we have $\tau_1 + \tau_2 + \tau_3 = 1$ by definition. For each $j \in \mathbb{N}$, we can apply Theorem 4.1 for $\mathrm{YQS}_{\mathbf{t}^{(j)}}^{w}$ and then consider the limiting behavior of the total costs for $j \to \infty$. (Letting the sample size go to infinity implies non-constant overhead per partitioning step for our implementation, which is not negligible any more. For the analysis here, we simply assume an oracle that provides us with the desired order statistic in constant time.)

For $\mathcal{H}(\mathbf{t}^{(j)})$, Equation (2) shows convergence to the entropy function $\mathcal{H}^* = \mathcal{H}^*(\boldsymbol{\tau}) = -\sum_{r=1}^{3} \tau_r \ln(\tau_r)$ and for the numerators a_C, a_S, a_{BC} and a_{SE}, it is easily seen that

$$\begin{aligned}
a_C^{(j)} &\to a_C^* := 1 + \tau_2 + (2\tau_1 + \tau_2)\tau_3\,, \\
a_S^{(j)} &\to a_S^* := \tau_1 + (\tau_1 + \tau_2)\tau_3\,, \\
a_{BC}^{(j)} &\to a_{BC}^* := 10 + 13\tau_1 + 5\tau_2 + (\tau_1 + \tau_2)(\tau_1 + 11\tau_3)\,, \\
a_{SE}^{(j)} &\to a_{SE}^* := 1 + \tau_1\,.
\end{aligned}$$



(a) $a^*_C/\mathcal{H}^*$ (b) $a^*_S/\mathcal{H}^*$

(c) $a^*_{BC}/\mathcal{H}^*$ (d) $a^*_{SE}/\mathcal{H}^*$

Figure 10 Contour plots for the limits of the leading-term coefficient of the overall number of comparisons, swaps, executed Bytecode instructions and scanned elements, as functions of $\boldsymbol{\tau}$. τ_1 and τ_2 are given on x- and y-axis, respectively, which determine τ_3 as $1 - \tau_1 - \tau_2$. Black dots mark global minima, white dots show the center point $\tau_1 = \tau_2 = \tau_3 = \frac{1}{3}$. (For swaps no minimum is attained in the open simplex, see main text). Black dashed lines are level lines connecting "equi-cost-ant" points, i.e., points of equal costs. White dotted lines mark points of equal entropy $\mathcal{H}^*$.

Together, the overall number of comparisons, swaps, Bytecodes and scanned elements converge to $a^*_C/\mathcal{H}^*$, $a^*_S/\mathcal{H}^*$, $a^*_{BC}/\mathcal{H}^*$ resp. $a^*_{SE}/\mathcal{H}^*$; see Figure 10 for plots of the four as functions in τ_1 and τ_2. We could not find a way to compute the minima of these functions analytically. However, all three functions have isolated minima that can be approximated well by numerical methods.

The number of comparisons is minimized for

$$\boldsymbol{\tau}^*_C \approx (0.428846, 0.268774, 0.302380) .$$

For this choice, the expected number of comparisons is asymptotically $1.4931\, n \ln n$. For swaps, the minimum is not attained inside the open simplex, but for the extreme points $\boldsymbol{\tau}^*_S = (0, 0, 1)$ and $\boldsymbol{\tau}^{*\prime}_S = (0, 1, 0)$. The minimal value of the coefficient is 0, so

the expected number of swaps drops to $o(n \ln n)$ for these extreme points. Of course, this is a very bad choice w. r. t. other cost measures, e. g., the number of comparisons becomes quadratic, which again shows the limitations of tuning an algorithm to one of its basic operations in isolation. The minimal asymptotic number of executed Bytecodes of roughly $16.3833\, n \ln n$ is obtained for

$$\boldsymbol{\tau}^*_{BC} \approx (0.206772, 0.348562, 0.444666) .$$

Finally, the least number of scanned elements, which is asymptotically $1.1346\, n \ln n$, is achieved for

$$\begin{aligned} \boldsymbol{\tau}^*_{SE} &= (q^2, q, q) \quad \text{with} \quad q = \sqrt{2} - 1 \\ &\approx (0.171573, 0.414214, 0.414214) . \end{aligned}$$

We note again that the optimal choices heavily differ depending on the employed cost measure and that the minima differ significantly from the symmetric choice $\boldsymbol{\tau} = (\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$.

8.4 Comparison with Classic Quicksort

8.4.1 Known Results for Classic Quicksort

Similarly to our Theorem 4.1, one can analyze the costs of classic Quicksort (CQS) with pivot sampling parameter $\mathbf{t} = (t_1, t_2) \in \mathbb{N}^2$, where the (single) pivot P is chosen as the $(t_1 + 1)$st-largest from a sample of $k = k(\mathbf{t}) = t_1 + t_2 + 1$ elements, see Martínez and Roura (2001). With $\mathcal{H}(t_1, t_2) := \sum_{r=1}^{2} \frac{t_r+1}{k+1}(H_{k+1} - H_{t_r+1})$ defined similarly as in Equation (1), we have the following results.

Theorem 8.1 (Expected Costs of CQS): *Generalized Classic Quicksort with pivot sampling parameter* $\mathbf{t} = (t_1, t_2)$ *performs on average* $C_n^{\mathrm{CQS}} \sim a_C^{\mathrm{CQS}}/\mathcal{H}\, n \ln n$ *comparisons,* $S_n^{\mathrm{CQS}} \sim a_S^{\mathrm{CQS}}/\mathcal{H}\, n \ln n$ *swaps and* $SE_n^{\mathrm{CQS}} \sim a_{SE}^{\mathrm{CQS}}/\mathcal{H}\, n \ln n$ *element scans to sort a random permutation of n elements, where*

$$\begin{aligned} a_C^{\mathrm{CQS}} &= a_{SE}^{\mathrm{CQS}} = 1 \qquad \textit{and} \\ a_S^{\mathrm{CQS}} &= \frac{(t_1 + 1)(t_2 + 1)}{(k + 1)(k + 2)} . \end{aligned}$$

Moreover, if the partitioning loop is implemented as in Listing 4 *of* (*Wild* 2012)*, it executes on average* $BC_n^{\mathrm{CQS}} \sim a_{BC}^{\mathrm{CQS}}/\mathcal{H}\, n \ln n$ *Java Bytecode instructions to sort a random permutation of size n with*

$$a_{BC}^{\mathrm{CQS}} = 6a_C^{\mathrm{CQS}} + 18a_S^{\mathrm{CQS}} . \qquad \square$$

Remark: In CQS, each element reached by a scanning index results in exactly one comparison (namely with the pivot). Therefore, the number of scanned elements and the number of key comparisons are exactly the same in CQS.

8.4.2 Pivots from Fixed Positions

The first theoretical studies of the new Quicksort variant invented by Yaroslavskiy assumed that pivots are chosen from fixed positions of the input. Trying to understand the reasons for its running time advantages we analyzed comparisons, swaps and the number of executed Bytecode instructions for YQS and CQS. However, comparing all related findings to corresponding results for classic Quicksort, we observed that YQS needs about 5% less comparisons than CQS, but performs about twice as many swaps, needs 65% more write accesses and executes about 20% more Bytecodes on average (Wild et al 2015). What is important here is that these results hold not only asymptotically, but already for practical n. (Without pivot sampling, an exact solution of the recurrences remains feasible.) Thus, it was somehow straightforward to utter the following conjecture.

Conjecture 5.1 of Wild et al (2015): *"The efficiency of Yaroslavskiy's algorithm in practice is caused by advanced features of modern processors. In models that assign constant cost contributions to single instructions — i. e., locality of memory accesses and instruction pipelining are ignored — classic Quicksort is more efficient."*

Kushagra et al (2014) then were the first to provide strong evidence for this conjecture by showing that YQS needs significantly less cache misses than CQS. Very recently, we were able to exclude the effects of pipelined execution from the list of potential explanations; both algorithms CQS and YQS give rise to about the same number of branch misses on average, so their rollback costs cannot be responsible for the differences in running time (Martínez et al 2015).

In this paper we present a precise analysis of the number of scanned elements per partitioning step (cf. Lemma 5.4). Plugging this result into the precise solution of the dual-pivot Quicksort recurrence without pivot sampling, we get the precise total number of scanned elements:

- YQS scans $1.6n \ln(n) - 2.2425n + O(\log n)$ elements on average, while
- CQS needs $2n \ln(n) - 2.3045n + O(\log n)$ element scans on average.
  (Recall that scanned elements and comparisons coincide in CQS, so we can reuse results for comparisons, see e. g. (Sedgewick 1977).)

Both results are actually known precisely, but the sublinear terms are really negligible for reasonable input sizes.

Obviously, the number of scanned elements is significantly smaller in YQS than in CQS *for all* n. Accordingly, and in the light of all the results mentioned before, we assume our conjecture to be verified (for pivots taken from fixed positions): YQS is more efficient in practice than CQS because it needs less element scans and thus uses the memory hierarchy more efficiently.

Note that asymptotically, YQS needs 25% less element scans, but at the same time executes 20% more Bytecodes. In terms of practical running time, it seems plausible that both Bytecodes and scanned elements yield their share. In experiments conducted by one of the authors, YQS was 13% faster in Java and 10% faster in C++ (Wild 2012), which is not explained well by either cost measure in isolation.

One might assume that a sensible model for actual running time is a *linear combination* of Bytecodes and scans

$$Q \;=\; (1-\mu)\cdot BC + \mu \cdot SE$$

for an (unknown) parameter $\mu \in [0,1]$. Intuitively, μ is the relative importance of the number of scanned elements for total running time. Inserting the results for CQS and YQS and solving $Q^{\mathrm{CQS}}/Q^{\mathrm{YQS}} = 1.1$ for μ, we get $\mu \approx 0.95$. (The solution actually depends on n, so there is one solution for every input size. However, we get $0.93 \le \mu \le 0.96$ for all $n \ge 100$.)

This means — assuming the linear model is correct — that 95% of the running time of Quicksort are caused by element scans and only 5% by executed Bytecodes. Stated otherwise, a single scanned element is as costly as executing 20 Bytecode instructions.

8.4.3 Pivots from Samples of Size k

While the last section discussed the most elementary versions of CQS and YQS, we will now come back to the case where pivots are chosen from a sample. To compare the single-pivot CQS with the dual-pivot YQS, we need two pivot sampling parameters $\mathbf{t}$, which we here call $\mathbf{t}^{\mathrm{CQS}} \in \mathbb{N}^2$ and $\mathbf{t}^{\mathrm{YQS}} \in \mathbb{N}^3$, respectively. Of course, they potentially result in different sample sizes $k^{\mathrm{CQS}} = t_1^{\mathrm{CQS}} + t_2^{\mathrm{CQS}} + 1$ and $k^{\mathrm{YQS}} = t_1^{\mathrm{YQS}} + t_2^{\mathrm{YQS}} + t_3^{\mathrm{YQS}} + 2$.

Analytic results for general pivot sampling are only available as leading-term asymptotics, so we have to confine ourselves to the comparison of CQS and YQS on very large inputs. Still, we consider it unsound to compare, say, YQS with a sample size $k^{\mathrm{YQS}} = 100$ to CQS with sample size $k^{\mathrm{CQS}} = 3$, where one algorithm is allowed to use much more information about the input to make its decision for good pivot values than the other. Moreover, even though sample size analytically only affect the linear term of costs, the former would in practice spend a non-negligible amount of its running time sorting the large samples, whereas the latter knows its pivot after just three quick key comparisons. For a fair competition, we will thus keep the sample sizes in the same range.

Once the sample size is fixed, one can still choose different order statistics of the sample. As the optimal choices for YQS are so sensitive to the employed cost measure, we will first focus on choosing symmetric pivots, i.e., $\mathbf{t}^{\mathrm{CQS}} = (t^{\mathrm{CQS}}, t^{\mathrm{CQS}})$ and $\mathbf{t}^{\mathrm{YQS}} = (t^{\mathrm{YQS}}, t^{\mathrm{YQS}}, t^{\mathrm{YQS}})$, for integers t^{CQS} and t^{YQS}, such that the sample sizes are exactly the same. This effectively limits the allowable sample sizes to $k = 6\lambda - 1$ for integers $\lambda \ge 1$; Table 4 shows the results up to $\lambda = 4$.

As k increases, the algorithms improve in all cost measures, except for the number of swaps in CQS. The reason is that swaps profit from unbalanced pivots, which we make less likely by sampling (see (Martínez and Roura 2001) and (Wild 2012) for a more detailed discussion). Moreover, the (relative) ranking of the two algorithms w. r. t. each cost measure in isolation is the same for all sample sizes and thus similar to the case without sampling (see Section 8.4.2) — with a single exception: without sampling, YQS need 5% less comparisons than CQS, but for all values of k in Table 4, YQS actually needs 5% *more* comparisons! As soon as the variance of the ranks of

k	cost measure	classic Quicksort	Yaroslavskiy's Quicksort
no sampling	comparisons	2	1.9
	swaps	$0.\overline{3}$	0.6
	Bytecodes	18	21.7
	scanned elements	2	1.6
5	comparisons	1.6216	1.7043
	swaps	0.3475	0.5514
	Bytecodes	15.9846	19.2982
	scanned elements	1.6216	1.4035
11	comparisons	1.5309	1.6090
	swaps	0.3533	0.5280
	Bytecodes	15.5445	18.1269
	scanned elements	1.5309	1.3073
17	comparisons	1.5012	1.5779
	swaps	0.3555	0.5204
	Bytecodes	15.4069	17.7435
	scanned elements	1.5012	1.2758
23	comparisons	1.4864	1.5625
	swaps	0.3567	0.5166
	Bytecodes	15.3401	17.5535
	scanned elements	1.4864	1.2601

Table 4 Comparison of CQS and YQS whose pivots are chosen equidistantly from samples of the given sizes. All entries give the (approximate) leading-term coefficient of the asymptotic cost for the given cost measure. By $0.\overline{3}$ we mean the repeating decimal $0.333\ldots = \frac{1}{3}$.

pivots is reduced by sampling, the advantage of YQS to exploit skewed pivots to save comparisons through clever use of asymmetries in the code is no longer enough to beat CQS if the latter chooses its pivot as median of a sample of the same size. This remains true if we allow YQS to choose the order statistics that minimize the number of comparisons: we then get as leading-term coefficients of the number of comparisons 1.7043, 1.5848, 1.5554 and 1.5396 for $k = 5$, 11, 17 and 23, respectively, which still is significantly more than for CQS with median-of-k.

This is a quite important observation, as it shows that the number of key comparisons cannot be the reason for YQS's success in practice: for the library implementations, YQS has always been compared to CQS with pivot sampling, i.e., to an algorithm that needs *less* comparisons than YQS. To be precise, the Quicksort implementation used in Java 6 is the version of Bentley and McIlroy (1993) which uses the *"ninther"* as pivot: Take three samples of three elements each, pick the median of each of the samples and then make the median of the three medians our pivot. The expected number of key comparisons used by this algorithm has been computed by Durand (2003). The leading-term coefficient is $\frac{12600}{8027} \approx 1.5697$, ranking between CQS with median-of-seven and median-of-nine. The version of Yaroslavskiy's Quicksort used in Java 7 uses the tertiles-of-five as pivots and needs (asymptotically) $1.7043\, n \ln n$ comparisons.

Similarly, CQS needs less swaps and Bytecode instructions than YQS. If we, however, compare the same two algorithms in terms of the number of scanned elements

they need, YQS clearly wins with $1.4035\,n\ln n$ vs. $1.5697\,n\ln n$ in the asymptotic average. Even quantitatively, this offers a plausible explanation of running time differences: The Java 7 Quicksort saves 12% of the element scans over the version in Java 6, which roughly matches speedups observed in running time studies.

One should note at this point, however, that the library versions are *not* direct implementations of the basic partitioning algorithms as given in Algorithm 2 for YQS. For example, the variant of Bentley and McIlroy (1993) actually does a three-way partitioning to efficiently deal with inputs with many equal keys and the Java 7 version of YQS uses similar tweaks. The question, whether scanned elements (or cache misses) are the dominating factor in the running time of these algorithms, needs further study.

We conclude that also for the pivot sampling strategies employed in practice, YQS clearly outperforms CQS in the number of scanned elements. It is most likely that this more efficient use of the memory hierarchy makes YQS faster in practice.

9 Conclusion

In this paper, we give the precise leading-term asymptotic of the average costs of Quicksort with Yaroslavskiy's dual-pivot partitioning method and selection of pivots as arbitrary order statistics of a constant-size sample for a variety of different cost measures: the number of key comparisons and the number of swaps (as classically used for sorting algorithms), but also the number of executed Java Bytecode instructions and the number of scanned elements, a new cost measure that we introduce as simple model for the number of cache misses.

The inherent asymmetries in Yaroslavskiy's partitioning algorithm lead to the situation that the symmetric choice for pivots, the tertiles of the sample, is *not* optimal: a deliberate, well-dosed skew in pivot selection improves overall performance. For the optimal skew, we have to find a trade-off between several counteracting effects and the result is very sensitive to the employed cost measure. The precise analysis in this paper can provide valuable guidance in choosing the right sampling scheme.

Whereas cache misses are complicated in detail and machine-dependent, scanned elements are a precisely defined, abstract cost measure that is as elementary as key comparisons or swaps. At the same time, it provides a reasonable approximation for the number of incurred cache misses, and we show in particular that the number of scanned elements is well-suited to compare different Quicksort variants w. r. t. their efficiency in the external-memory model.

Comparing classic single-pivot Quicksort with Yaroslavskiy's dual-pivot Quicksort in terms of scanned elements finally yields a convincing analytical explanation why the latter is found to be more efficient in practice: Yaroslavskiy's algorithm needs much less element scans and thus uses the memory hierarchy more efficiently, with and without pivot sampling.

In light of the complexity of modern machines, it is implausible that a single simple cost measure captures all contributions to running time; rather, it seems likely that the number of scanned elements (memory accesses) and the number of executed instructions in the CPU both have significant influence. With algorithms as exces-

sively studied and tuned as Quicksort, we have reached a point where slight changes in the underlying hardware architecture can shift the weights of these factors enough to make variants of an algorithm superior on today's machines which were not competitive on yesterday's machines: CPU speed has increased much more than memory speed, shifting the weights towards algorithms that save in scanned elements, like Yaroslavskiy's dual-pivot Quicksort.

Future work. A natural extension of this work would be the computation of the linear term of costs, which is not negligible for moderate n. This will require a much more detailed analysis as sorting the samples and dealing with short subarrays contribute to the linear term of costs, but then allows to compute the optimal choice for w, as well. While in this paper only expected values were considered, the distributional analysis of Section 5 can be used as a starting point for analyzing the distribution of overall costs. Yaroslavskiy's partitioning can also be used in Quickselect (Wild et al 2014); the effects of generalized pivot sampling there are yet to be studied. Finally, other cost measures, like the number of symbol comparisons (Vallée et al 2009; Fill and Janson 2012), would be interesting to analyze.

Acknowledgements We thank two anonymous reviewers for their careful reading and helpful comments.

Bibliography

Aumüller M, Dietzfelbinger M (2013) Optimal partitioning for dual pivot quicksort. In: Fomin FV, Freivalds R, Kwiatkowska M, Peleg D (eds) International Colloquium on Automata, Languages and Programming, Springer, LNCS, vol 7965, pp 33–44

Bentley JL, McIlroy MD (1993) Engineering a sort function. Software: Practice and Experience 23(11):1249–1265

Chern HH, Hwang HK, Tsai TH (2002) An asymptotic theory for cauchy–euler differential equations with applications to the analysis of algorithms. Journal of Algorithms 44(1):177–225

Chung KL (2001) A Course in Probability Theory, 3rd edn. Academic Press

Cormen TH, Leiserson CE, Rivest RL, Stein C (2009) Introduction to Algorithms, 3rd edn. MIT Press

David HA, Nagaraja HN (2003) Order Statistics, 3rd edn. Wiley-Interscience

Durand M (2003) Asymptotic analysis of an optimized quicksort algorithm. Information Processing Letters 85(2):73–77

van Emden MH (1970) Increasing the efficiency of quicksort. Communications of the ACM 13(9):563–567

Estivill-Castro V, Wood D (1992) A survey of adaptive sorting algorithms. ACM Computing Surveys 24(4):441–476

Fill J, Janson S (2012) The number of bit comparisons used by quicksort: an average-case analysis. Electronic Journal of Probability 17:1–22

Graham RL, Knuth DE, Patashnik O (1994) Concrete Mathematics: A Foundation for Computer Science. Addison-Wesley

Hennequin P (1991) Analyse en moyenne d'algorithmes : tri rapide et arbres de recherche. PhD Thesis, Ecole Politechnique, Palaiseau

Hennessy JL, Patterson DA (2006) Computer Architecture: A Quantitative Approach, 4th edn. Morgan Kaufmann Publishers

Hoare CAR (1961) Algorithm 65: Find. Communications of the ACM 4(7):321–322

Kaligosi K, Sanders P (2006) How branch mispredictions affect quicksort. In: Erlebach T, Azar Y (eds) European Symposium on Algorithms, Springer, LNCS, vol 4168, pp 780–791

Kushagra S, López-Ortiz A, Qiao A, Munro JI (2014) Multi-pivot quicksort: Theory and experiments. In: McGeoch CC, Meyer U (eds) Meeting on Algorithm Engineering and Experiments, SIAM, pp 47–60

LaMarca A, Ladner RE (1999) The influence of caches on the performance of sorting. Journal of Algorithms 31(1):66–104
Mahmoud HM (2000) Sorting: A Distribution Theory. John Wiley & Sons
Martínez C, Roura S (2001) Optimal sampling strategies in quicksort and quickselect. SIAM Journal on Computing 31(3):683–705
Martínez C, Nebel ME, Wild S (2015) Analysis of branch misses in quicksort. In: Sedgewick R, Ward MD (eds) Meeting on Analytic Algorithmics and Combinatorics, SIAM, pp 114–128
Musser DR (1997) Introspective sorting and selection algorithms. Software: Practice and Experience 27(8):983–993
Nebel ME, Wild S (2014) Pivot sampling in dual-pivot quicksort. In: Bousquet-Mélou M, Soria M (eds) International Conference on Probabilistic, Combinatorial and Asymptotic Methods for the Analysis of Algorithms, DMTCS-HAL Proceedings Series, vol BA, pp 325–338
Neininger R (2001) On a multivariate contraction method for random recursive structures with applications to quicksort. Random Structures & Algorithms 19(3-4):498–524
Roura S (2001) Improved master theorems for divide-and-conquer recurrences. Journal of the ACM 48(2):170–205
Sedgewick R (1975) Quicksort. PhD Thesis, Stanford University
Sedgewick R (1977) The analysis of quicksort programs. Acta Informatica 7(4):327–355
Sedgewick R (1978) Implementing quicksort programs. Communications of the ACM 21(10):847–857
Vallée B, Clément J, Fill JA, Flajolet P (2009) The number of symbol comparisons in quicksort and quickselect. In: Albers S, Marchetti-Spaccamela A, Matias Y, Nikoletseas S, Thomas W (eds) International Colloquium on Automata, Languages and Programming, Springer, LNCS, vol 5555, pp 750–763
Wild S (2012) Java 7's Dual-Pivot Quicksort. Master thesis, University of Kaiserslautern
Wild S, Nebel ME (2012) Average case analysis of Java 7's dual pivot quicksort. In: Epstein L, Ferragina P (eds) European Symposium on Algorithms, Springer, LNCS, vol 7501, pp 825–836
Wild S, Nebel ME, Reitzig R, Laube U (2013) Engineering Java 7's dual pivot quicksort using MaLiJAn. In: Sanders P, Zeh N (eds) Meeting on Algorithm Engineering and Experiments, SIAM, pp 55–69
Wild S, Nebel ME, Mahmoud H (2014) Analysis of quickselect under Yaroslavskiy's dual-pivoting algorithm. Algorithmica (to appear), DOI 10.1007/s00453-014-9953-x
Wild S, Nebel ME, Neininger R (2015) Average case and distributional analysis of Java 7's dual pivot quicksort. ACM Transactions on Algorithms 11(3):22:1–22:42

Appendix

A Index of Used Notation

In this section, we collect the notations used in this paper. (Some might be seen as "standard", but we think including them here hurts less than a potential misunderstanding caused by omitting them.)

Generic Mathematical Notation

$0.\overline{3}$ . . . . . repeating decimal; $0.\overline{3} = 0.333\ldots = \frac{1}{3}$.
The numerals under the line form the repeated part of the decimal number.

$\ln n$ . . . . . natural logarithm.

linearithmic . . . . . A function is "linearithmic" if it has order of growth $\Theta(n \log n)$.

$\mathbf{x}$ . . . . . to emphasize that $\mathbf{x}$ is a vector, it is written in **bold**;
components of the vector are not written in bold: $\mathbf{x} = (x_1, \ldots, x_d)$.

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

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

$\mathrm{Dir}(\boldsymbol{\alpha})$ . . . . . Dirichlet distributed random variable, $\boldsymbol{\alpha} \in \mathbb{R}^d_{>0}$.

$\mathrm{Mult}(n, \mathbf{p})$ . . . . . multinomially distributed random variable; $n \in \mathbb{N}$ and $\mathbf{p} \in [0, 1]^d$ with $\sum_{i=1}^{d} p_i = 1$.

$\mathrm{HypG}(k, r, n)$ . . . . . hypergeometrically distributed random variable; $n \in \mathbb{N}$, $k, r, \in \{1, \ldots, n\}$.

$\mathrm{B}(p)$ . . . . . Bernoulli distributed random variable; $p \in [0, 1]$.

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

$\mathrm{B}(\alpha_1, \ldots, \alpha_d)$ . . . . d-dimensional Beta function; defined in Equation (12) (page 44).

$\mathbb{E}[X]$ . . . . . expected value of X; we write $\mathbb{E}[X \mid Y]$ for the conditional expectation of X given Y.

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

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

$X_{(i)}$ . . . . . ith order statistic of a set of random variables $X_1, \ldots, X_n$,
i.e., the ith smallest element of $X_1, \ldots, X_n$.

$\mathbb{1}_{\{E\}}$ . . . . . indicator variable for event E, i.e., $\mathbb{1}_{\{E\}}$ is 1 if E occurs and 0 otherwise.

$a^{\underline{b}}$, $a^{\overline{b}}$ . . . . . factorial powers notation of Graham et al (1994); "a to the b falling resp. rising".

Input to the Algorithm

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

$\mathtt{A}$ . . . . . input array containing the items $\mathtt{A}[1], \ldots, \mathtt{A}[n]$ to be sorted; initially, $\mathtt{A}[i] = U_i$.

U_i . . . . . ith element of the input, i.e., initially $\mathtt{A}[i] = U_i$.
We assume $U_1, \ldots, U_n$ are i. i. d. $\mathcal{U}(0, 1)$ distributed.

Notation Specific to the Algorithm

$\mathbf{t} \in \mathbb{N}^3$ . . . . . pivot sampling parameter, see Section 3.1 (page 5).

$k = k(\mathbf{t})$ . . . . . sample size; defined in terms of $\mathbf{t}$ as $k(\mathbf{t}) = t_1 + t_2 + t_3 + 2$.

w . . . . . Insertionsort threshold; for $n \le w$, Quicksort recursion is truncated and we sort the subarray by Insertionsort.

M . . . . . cache size; the number of array elements that fit into the idealized cache; we assume $M \ge B$, $B \mid M$ (M is a multiple of B) and $B \mid n$; see Section 7.2.

B . . . . . block size; the number of array elements that fit into one cache block/line; see also M.

YQS, $\text{YQS}_{\mathbf{t}}^{w}$ . . . . . abbreviation for dual-pivot Quicksort with Yaroslavskiy's partitioning method, where pivots are chosen by generalized pivot sampling with parameter $\mathbf{t}$ and where we switch to Insertionsort for subproblems of size at most w.

CQS . . . . . abbreviation for classic (single-pivot) Quicksort using Hoare's partitioning, see e.g. (Sedgewick 1977, p. 329); a variety of notations are with CQS in the superscript to denote the corresponding quantities for classic Quicksort, e.g., C_n^{CQS} is the number of (partitioning) comparisons needed by CQS on a random permutation of size n.

$\mathbf{V} \in \mathbb{N}^k$ . . . . . (random) sample for choosing pivots in the first partitioning step.

P, Q . . . . . (random) values of chosen pivots in the first partitioning step.

small element . . . . . element U is small if $U < P$.

medium element . . . . element U is medium if $P < U < Q$.

large element . . . . . element U is large if $Q < U$.

sampled-out element . . the $k-2$ elements of the sample that are *not* chosen as pivots.

ordinary element . . . . the $n-k$ elements that have not been part of the sample.

k, g, ℓ . . . . . index variables used in Yaroslavskiy's partitioning method, see Algorithm 1 (page 7).

$\mathcal{K}, \mathcal{G}, \mathcal{L}$ . . . . . set of all (index) values attained by pointers k, g resp. ℓ during the first partitioning step; see Section 3.2 (page 6) and proof of Lemma 5.1 (page 17).

$c\,@\,\mathcal{P}$ . . . . . $c \in \{s, m, l\}$, $\mathcal{P} \subset \{1, \ldots, n\}$
(random) number of c-type ($s$mall, $m$edium or $l$arge) elements that are initially located at positions in $\mathcal{P}$, i.e., $c\,@\,\mathcal{P} \;=\; |\{i \in \mathcal{P} : U_i \text{ has type } c\}|$.

$l\,@\,\mathcal{K}$, $s\,@\,\mathcal{K}$, $s\,@\,\mathcal{G}$ . . see $c\,@\,\mathcal{P}$

χ . . . . . (random) point where k and g first meet.

δ . . . . . indicator variable of the random event that χ is on a large element, i.e., $\delta = \mathbb{1}_{\{U_\chi > Q\}}$.

$C_n^{\texttt{type}}$ . . . . . with $\texttt{type} \in \{\texttt{root}, \texttt{left}, \texttt{middle}, \texttt{right}\}$; (random) costs of a (recursive) call to GeneralizedYaroslavskiy(A, *left*, *right*, type) where A[*left*..*right*] contains n elements, i.e., $right - left + 1 = n$. The array elements are assumed to be in random order, except for the t_1, resp. t_2 leftmost elements for $C_n^{\texttt{left}}$ and $C_n^{\texttt{middle}}$ and the t_3 rightmost elements for $C_n^{\texttt{right}}$;
for all types holds $C_n^{\texttt{type}} \overset{\mathcal{D}}{=} C_n + O(n)$, see Section 5.1.

$T_n^{\texttt{type}}$ . . . . . with $\texttt{type} \in \{\texttt{root}, \texttt{left}, \texttt{middle}, \texttt{right}\}$; the costs of the first partitioning step of a call to GeneralizedYaroslavskiy(A, *left*, *right*, type); for all types holds $T_n^{\texttt{type}} \overset{\mathcal{D}}{=} T_n + O(1)$, see Section 5.1.

T_n . . . . . the costs of the first partitioning step, where *only* costs of procedure Partition are counted, see Section 5.1.

$W_n^{\texttt{type}}$ . . . . . with $\texttt{type} \in \{\texttt{root}, \texttt{left}, \texttt{middle}, \texttt{right}\}$; as $C_n^{\texttt{type}}$, but the calls are InsertionSortLeft(A, *left*, *right*, 1) for $W_n^{\texttt{root}}$,
InsertionSortLeft(A, *left*, *right*, $\max\{t_1, 1\}$) for $W_n^{\texttt{left}}$
InsertionSortLeft(A, *left*, *right*, $\max\{t_2, 1\}$) for $W_n^{\texttt{middle}}$ and
InsertionSortRight(A, *left*, *right*, $\max\{t_3, 1\}$) for $W_n^{\texttt{right}}$.

W_n . . . . . (random) costs of sorting a random permutation of size n with Insertionsort.

C_n, S_n, BC_n, SE_n . . (random) number of comparisons / swaps / Bytecodes / scanned elements of $\text{YQS}_{\mathbf{t}}^{w}$ on a random permutation of size n that are caused in procedure Partition; see Section 1.1 for more information on the cost measures; in Section 5.1, C_n is used as general placeholder for any of the above cost measures.

T_C, T_S, T_{BC}, T_{SE} . . . . (random) number of comparisons / swaps / Bytecodes / element scans of the first partitioning step of $\text{YQS}_{\mathbf{t}}^{w}$ on a random permutation of size n;
$T_C(n)$, $T_S(n)$ and $T_{BC}(n)$ when we want to emphasize dependence on n.

a_C, a_S, a_{BC}, a_{SE} . . . coefficient of the linear term of $\mathbb{E}[T_C(n)]$, $\mathbb{E}[T_S(n)]$, $\mathbb{E}[T_{BC}(n)]$ and $\mathbb{E}[T_{SE}(n)]$; see Theorem 4.1 (page 13).

$\mathcal{H}$ . . . . . discrete entropy; defined in Equation (1) (page 13).

$\mathcal{H}^*(\mathbf{p})$ . . . . . continuous (Shannon) entropy with basis e; defined in Equation (2) (page 13).

$\mathbf{J} \in \mathbb{N}^3$ . . . . . (random) vector of subproblem sizes for recursive calls;
for initial size n, we have $\mathbf{J} \in \{0, \ldots, n-2\}^3$ with $J_1 + J_2 + J_3 = n - 2$.

$\mathbf{I} \in \mathbb{N}^3$ . . . . . (random) vector of partition sizes, i.e., the number of small, medium resp. large *ordinary* elements; for initial size n, we have $\mathbf{I} \in \{0, \ldots, n-k\}^3$ with $I_1 + I_2 + I_3 = n - k$;
$\mathbf{J} = \mathbf{I} + \mathbf{t}$ and conditional on $\mathbf{D}$ we have $\mathbf{I} \overset{\mathcal{D}}{=} \mathrm{Mult}(n-k, \mathbf{D})$.

$\mathbf{D} \in [0,1]^3$ . . . . . (random) spacings of the unit interval $(0,1)$ induced by the pivots P and Q, i.e., $\mathbf{D} = (P, Q-P, 1-Q)$; $\mathbf{D} \overset{\mathcal{D}}{=} \mathrm{Dir}(\mathbf{t}+1)$.

$a^*_C, a^*_S, a^*_{BC}, a^*_{SE}$ . . . limit of a_C, a_S, a_{BC} resp. a_{SE} for the optimal sampling parameter $\mathbf{t}$ when $k \to \infty$.

$\boldsymbol{\tau}^*_C, \boldsymbol{\tau}^*_S, \boldsymbol{\tau}^*_{BC}, \boldsymbol{\tau}^*_{SE}$ . . . optimal limiting ratio $\mathbf{t}/k \to \boldsymbol{\tau}^*_C$ such that $a_C \to a^*_C$ (resp. for S, BC and SE).

B Properties of Distributions

We herein collect definitions and basic properties of the distributions used in this paper. They will be needed for computing expected values in Appendix C. This appendix is an update of Appendix C in (Nebel and Wild 2014), which we include here for the reader's convenience.

We use the notation $x^{\overline{n}}$ and $x^{\underline{n}}$ of Graham et al (1994) for rising and falling factorial powers, respectively.

B.1 Dirichlet Distribution and Beta Function

For $d \in \mathbb{N}$ let Δ_d be the standard $(d-1)$-dimensional simplex, i.e.,

$$\Delta_d \;:=\; \left\{ x = (x_1, \ldots, x_d) \;:\; \forall i : x_i \geq 0 \;\wedge \sum_{1 \leq i \leq d} x_i = 1 \right\}. \tag{10}$$

Let $\alpha_1, \ldots, \alpha_d > 0$ be positive reals. A random variable $\mathbf{X} \in \mathbb{R}^d$ is said to have the *Dirichlet distribution* with *shape parameter* $\boldsymbol{\alpha} := (\alpha_1, \ldots, \alpha_d)$ — abbreviated as $\mathbf{X} \stackrel{\mathcal{D}}{=} \mathrm{Dir}(\boldsymbol{\alpha})$ — if it has a density given by

$$f_{\mathbf{X}}(x_1, \ldots, x_d) \;:=\; \begin{cases} \frac{1}{\mathrm{B}(\boldsymbol{\alpha})} \cdot x_1^{\alpha_1 - 1} \cdots x_d^{\alpha_d - 1}, & \text{if } \mathbf{x} \in \Delta_d\,; \\ 0, & \text{otherwise}. \end{cases} \tag{11}$$

Here, $\mathrm{B}(\boldsymbol{\alpha})$ is the *d-dimensional Beta function* defined as the following Lebesgue integral:

$$\mathrm{B}(\alpha_1, \ldots, \alpha_d) \;:=\; \int_{\Delta_d} x_1^{\alpha_1 - 1} \cdots x_d^{\alpha_d - 1} \, \mu(d\mathbf{x}) \,. \tag{12}$$

The integrand is exactly the density without the normalization constant $\frac{1}{\mathrm{B}(\boldsymbol{\alpha})}$, hence $\int f_X \, d\mu = 1$ as needed for probability distributions.

The Beta function can be written in terms of the Gamma function $\Gamma(t) = \int_0^\infty x^{t-1} e^{-x} \, dx$ as

$$\mathrm{B}(\alpha_1, \ldots, \alpha_d) \;=\; \frac{\Gamma(\alpha_1) \cdots \Gamma(\alpha_d)}{\Gamma(\alpha_1 + \cdots + \alpha_d)} \,. \tag{13}$$

(For integral parameters $\boldsymbol{\alpha}$, a simple inductive argument and partial integration suffice to prove (13).)
Note that $\mathrm{Dir}(1, \ldots, 1)$ corresponds to the uniform distribution over Δ_d. For integral parameters $\boldsymbol{\alpha} \in \mathbb{N}^d$, $\mathrm{Dir}(\boldsymbol{\alpha})$ is the distribution of the *spacings* or *consecutive differences* induced by appropriate order statistics of i. i. d. uniformly in $(0, 1)$ distributed random variables, as summarized in the following proposition.

Proposition B.1 (David and Nagaraja 2003, Section 6.4): *Let $\boldsymbol{\alpha} \in \mathbb{N}^d$ be a vector of positive integers and set $k := -1 + \sum_{i=1}^d \alpha_i$. Further let $V_1, \ldots, V_k$ be k random variables i. i. d. uniformly in $(0, 1)$ distributed. Denote by $V_{(1)} \leq \cdots \leq V_{(k)}$ their corresponding order statistics. We select some of the order statistics according to $\boldsymbol{\alpha}$: for $j = 1, \ldots, d-1$ define $W_j := V_{(p_j)}$, where $p_j := \sum_{i=1}^j \alpha_i$. Additionally, we set $W_0 := 0$ and $W_d := 1$.*

Then, the consecutive distances (or spacings) $D_j := W_j - W_{j-1}$ for $j = 1, \ldots, d$ induced by the selected order statistics $W_1, \ldots, W_{d-1}$ are Dirichlet distributed with parameter $\boldsymbol{\alpha}$:

$$(D_1, \ldots, D_d) \;\stackrel{\mathcal{D}}{=}\; \mathrm{Dir}(\alpha_1, \ldots, \alpha_d) \,. \qquad \square$$

In the computations of Section 6.1, mixed moments of Dirichlet distributed variables will show up, which can be dealt with using the following general statement.

Lemma B.2: *Let $\mathbf{X} = (X_1, \ldots, X_d) \in \mathbb{R}^d$ be a $\mathrm{Dir}(\boldsymbol{\alpha})$ distributed random variable with parameter $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_d)$. Let further $m_1, \ldots, m_d \in \mathbb{N}$ be non-negative integers and abbreviate the sums $A := \sum_{i=1}^d \alpha_i$ and $M := \sum_{i=1}^d m_i$. Then we have*

$$\mathbb{E}[X_1^{m_1} \cdots X_d^{m_d}] \;=\; \frac{\alpha_1^{\overline{m_1}} \cdots \alpha_d^{\overline{m_d}}}{A^{\overline{M}}} \,.$$

Proof: Using $\frac{\Gamma(z+n)}{\Gamma(z)} = z^{\overline{n}}$ for all $z \in \mathbb{R}_{>0}$ and $n \in \mathbb{N}$, we compute

$$\begin{aligned}
\mathbb{E}[X_1^{m_1} \cdots X_d^{m_d}] &= \int_{\Delta_d} x_1^{m_1} \cdots x_d^{m_d} \cdot \frac{x_1^{\alpha_1-1} \cdots x_d^{\alpha_d-1}}{\mathrm{B}(\boldsymbol{\alpha})} \, \mu(dx) && (14)\\
&= \frac{\mathrm{B}(\alpha_1 + m_1, \ldots, \alpha_d + m_d)}{\mathrm{B}(\alpha_1, \ldots, \alpha_d)} && (15)\\
&\underset{(13)}{=} \frac{\alpha_1^{\overline{m_1}} \cdots \alpha_d^{\overline{m_d}}}{A^{\overline{M}}} \, . && (16)
\end{aligned}$$

□

For completeness, we state here a two-dimensional Beta integral with an additional logarithmic factor that is needed in Appendix D (see also Martínez and Roura 2001, Appendix B):

$$\begin{aligned}
\mathrm{B}_{\ln}(\alpha_1, \alpha_2) &:= -\int_0^1 x^{\alpha_1-1}(1-x)^{\alpha_2-1} \ln x \, dx \\
&= \mathrm{B}(\alpha_1, \alpha_2)(H_{\alpha_1+\alpha_2-1} - H_{\alpha_1-1}) \, . && (17)
\end{aligned}$$

For integral parameters $\boldsymbol{\alpha}$, the proof is elementary: By partial integration, we can find a recurrence equation for $\mathrm{B}_{\ln}$:

$$\mathrm{B}_{\ln}(\alpha_1, \alpha_2) = \frac{1}{\alpha_1}\mathrm{B}(\alpha_1, \alpha_2) + \frac{\alpha_2 - 1}{\alpha_1}\mathrm{B}_{\ln}(\alpha_1 + 1, \alpha_2 - 1) \, .$$

Iterating this recurrence until we reach the base case $\mathrm{B}_{\ln}(a, 0) = \frac{1}{a^2}$ and using (13) to expand the Beta function, we obtain (17).

B.2 Multinomial Distribution

Let $n, d \in \mathbb{N}$ and $k_1, \ldots, k_d \in \mathbb{N}$. *Multinomial coefficients* are the multidimensional extension of binomials:

$$\binom{n}{k_1, k_2, \ldots, k_d} := \begin{cases} \dfrac{n!}{k_1!k_2!\cdots k_d!}, & \text{if } n = \sum_{i=1}^{d} k_i \, ; \\ 0, & \text{otherwise.} \end{cases}$$

Combinatorially, $\binom{n}{k_1,\ldots,k_d}$ is the number of ways to partition a set of n objects into d subsets of respective sizes $k_1, \ldots, k_d$ and thus they appear naturally in the *multinomial theorem*:

$$(x_1 + \cdots + x_d)^n = \sum_{\substack{i_1,\ldots,i_d \in \mathbb{N} \\ i_1+\cdots+i_d=n}} \binom{n}{i_1, \ldots, i_d} x_1^{i_1} \cdots x_d^{i_d} \qquad \text{for } n \in \mathbb{N} \, . \tag{18}$$

Let $p_1, \ldots, p_d \in [0, 1]$ such that $\sum_{i=1}^{d} p_i = 1$. A random variable $\mathbf{X} \in \mathbb{N}^d$ is said to have *multinomial distribution* with parameters n and $\mathbf{p} = (p_1, \ldots, p_d)$ — written shortly as $\mathbf{X} \overset{\mathcal{D}}{=} \mathrm{Mult}(n, \mathbf{p})$ — if for any $\mathbf{i} = (i_1, \ldots, i_d) \in \mathbb{N}^d$ holds

$$\mathbb{P}(\mathbf{X} = \mathbf{i}) = \binom{n}{i_1, \ldots, i_d} p_1^{i_1} \cdots p_d^{i_d} \, .$$

We need some expected values involving multinomial variables. They can be expressed as special cases of the following mixed factorial moments.

Lemma B.3: *Let $p_1, \ldots, p_d \in [0, 1]$ such that $\sum_{i=1}^{d} p_i = 1$ and consider a $\mathrm{Mult}(n, \mathbf{p})$ distributed variable $\mathbf{X} = (X_1, \ldots, X_d) \in \mathbb{N}^d$. Let further $m_1, \ldots, m_d \in \mathbb{N}$ be non-negative integers and abbreviate their sum as $M := \sum_{i=1}^{d} m_i$. Then we have*

$$\mathbb{E}[(X_1)^{\underline{m_1}} \cdots (X_d)^{\underline{m_d}}] = n^{\underline{M}} p_1^{m_1} \cdots p_d^{m_d} \, .$$

Proof: We compute

$$\begin{aligned}
\mathbb{E}[(X_1)^{\underline{m_1}} \cdots (X_d)^{\underline{m_d}}] &= \sum_{\mathbf{x}\in\mathbb{N}^d} x_1^{\underline{m_1}} \cdots x_d^{\underline{m_d}} \binom{n}{x_1, \ldots, x_d} p_1^{x_1} \cdots p_d^{x_d} \\
&= n^{\underline{M}} p_1^{m_1} \cdots p_d^{m_d} \times \\
&\quad \sum_{\substack{\mathbf{x}\in\mathbb{N}^d: \\ \forall i: x_i \geq m_i}} \binom{n-M}{x_1 - m_1, \ldots, x_d - m_d} p_1^{x_1 - m_1} \cdots p_d^{x_d - m_d} \\
&\underset{(18)}{=} n^{\underline{M}} p_1^{m_1} \cdots p_d^{m_d} \ (\underbrace{p_1 + \cdots + p_d}_{=1})^{n-M} \\
&= n^{\underline{M}} p_1^{m_1} \cdots p_d^{m_d} \ . \qquad (19)
\end{aligned}$$

□

C Proof of Lemma 6.1

In this appendix, we give the computations needed to prove Lemma 6.1. They were also given in Appendix D of (Nebel and Wild 2014), but we reproduce them here for the reader's convenience.

We recall that $\mathbf{D} \overset{\mathcal{D}}{=} \mathrm{Dir}(\mathbf{t}+1)$ and $\mathbf{I} \overset{\mathcal{D}}{=} \mathrm{Mult}(n-k, \mathbf{D})$ and start with the simple ingredients: $\mathbb{E}[I_j]$ for $j = 1, 2, 3$.

$$\begin{aligned}
\mathbb{E}[I_j] &= \mathbb{E}_{\mathbf{D}}[\mathbb{E}[I_j \mid \mathbf{D} = \mathbf{d}]] \\
&\underset{\text{Lemma B.3}}{=} \mathbb{E}_{\mathbf{D}}[D_j(n-k)] \\
&\underset{\text{Lemma B.2}}{=} (n-k)\frac{t_j+1}{k+1} \ . \qquad (20)
\end{aligned}$$

The term $\mathbb{E}[\mathrm{B}(\frac{I_3}{n-k})]$ is then easily computed using (20):

$$\mathbb{E}[\mathrm{B}(\tfrac{I_3}{n-k})] = \frac{\mathbb{E}[I_3]}{n-k} = \frac{t_3+1}{k+1} = \Theta(1) \ . \qquad (21)$$

This leaves us with the hypergeometric variables; using the well-known formula $\mathbb{E}[\mathrm{HypG}(k, r, n)] = k\frac{r}{n}$, we find

$$\begin{aligned}
\mathbb{E}[\mathrm{HypG}(I_1 + I_2, I_3, n-k)] &= \mathbb{E}_{\mathbf{I}}\Big[\mathbb{E}[\mathrm{HypG}(i_1 + i_2, i_3, n-k) \mid \mathbf{I} = \mathbf{i}]\Big] \\
&= \mathbb{E}\left[\frac{(I_1 + I_2)I_3}{n-k}\right] \\
&= \mathbb{E}_{\mathbf{D}}\left[\frac{\mathbb{E}[I_1 I_3 \mid \mathbf{D}] + \mathbb{E}[I_2 I_3 \mid \mathbf{D}]}{n-k}\right] \\
&\underset{\text{Lemma B.3}}{=} \frac{(n-k)^{\underline{2}}\mathbb{E}[D_1 D_3] + (n-k)^{\underline{2}}\mathbb{E}[D_2 D_3]}{n-k} \\
&\underset{\text{Lemma B.2}}{=} \frac{((t_1+1)+(t_2+1))(t_3+1)}{(k+1)^{\overline{2}}}(n-k-1) \ . \qquad (22)
\end{aligned}$$

The second hypergeometric summand is obtained similarly. □

D Solution to the Recurrence

This appendix is an update of Appendix E in (Nebel and Wild 2014), we include it here for the reader's convenience.

An elementary proof can be given for Theorem 6.2 using Roura's *Continuous Master Theorem* (CMT) (Roura 2001). The CMT applies to a wide class of full-history recurrences whose coefficients can be well-approximated asymptotically by a so-called *shape function* $w : [0, 1] \to \mathbb{R}$. The shape function describes the coefficients only depending on the *ratio* j/n of the subproblem size j and the current size n (not depending on n or j itself) and it smoothly continues their behavior to any real number $z \in [0, 1]$. This continuous point of view also allows to compute precise asymptotics for complex discrete recurrences via fairly simple integrals.

Theorem D.1 (Martínez and Roura 2001, Theorem 18): *Let F_n be recursively defined by*

$$F_n \;=\; \begin{cases} b_n, & \text{for } 0 \le n < N; \\ t_n \;+\; \sum_{j=0}^{n-1} w_{n,j}\, F_j, & \text{for } n \ge N \end{cases} \tag{23}$$

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

$$\sum_{j=0}^{n-1} \left| w_{n,j} \;-\; \int_{j/n}^{(j+1)/n} w(z)\, dz \right| \;=\; O(n^{-d}), \qquad (n \to \infty), \tag{24}$$

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

1. *If $H > 0$, then $F_n \sim \dfrac{t_n}{H}$.*
2. *If $H = 0$, then $F_n \sim \dfrac{t_n \ln n}{\tilde{H}}$ with $\tilde{H} = -(\beta + 1) \int_0^1 z^\alpha \ln(z)\, w(z)\, dz$.*
3. *If $H < 0$, then $F_n \sim \Theta(n^c)$ for the unique $c \in \mathbb{R}$ with $\int_0^1 z^c w(z)\, dz = 1$.* □

The analysis of single-pivot Quicksort with pivot sampling is the application par excellence for the CMT (Martínez and Roura 2001). We will generalize this work of Martínez and Roura to the dual-pivot case.

Note that the recurrence for F_n depends *linearly* on t_n, so whenever $t_n = t'_n + t''_n$, we can apply the CMT to both the summands of the toll function separately and sum up the results. In particular, if we have an asymptotic expansion for t_n, we get an asymptotic expansion for F_n; the latter might however get truncated in precision when we end up in case 3 of Theorem D.1.

Our Equation (6) on page 21 has the form of (23) with

$$w_{n,j} \;=\; \sum_{r=1}^{3} \mathbb{P}(J_r = j)\ .$$

Recall that $\mathbf{J} = \mathbf{I} + \mathbf{t}$ and that $\mathbf{I} \overset{\mathcal{D}}{=} \mathrm{Mult}(n - k, \mathbf{D})$ conditional on $\mathbf{D}$, which in turn is a random variable with distribution $\mathbf{D} \overset{\mathcal{D}}{=} \mathrm{Dir}(\mathbf{t} + 1)$.

The probabilities $\mathbb{P}(J_r = j) = \mathbb{P}(I_r = j - t_r)$ can be computed using that the marginal distribution of I_r is binomial $\mathrm{Bin}(N, D_r)$, where we abbreviate by $N := n - k$ the number of ordinary elements. It is convenient to consider $\tilde{\mathbf{D}} := (D_r, 1 - D_r)$, which is distributed like $\tilde{\mathbf{D}} \overset{\mathcal{D}}{=} \mathrm{Dir}(t_r + 1, k - t_r)$. For $i \in [0..N]$ holds

$$\begin{aligned} \mathbb{P}(I_r = i) \;&=\; \mathbb{E}_{\mathbf{D}}\left[\mathbb{E}_{\mathbf{J}}[\mathbb{1}_{\{I_r = i\}} \mid \mathbf{D}]\right] \\ &=\; \mathbb{E}_{\mathbf{D}}\left[\tbinom{N}{i} \tilde{D}_1^i \tilde{D}_2^{N-i}\right] \\ &\overset{\text{Lemma B.2}}{=}\; \binom{N}{i} \frac{(t_r + 1)^{\overline{i}}\,(k - t_r)^{\overline{N-i}}}{(k+1)^{\overline{N}}}\ . \end{aligned} \tag{25}$$

D.1 Finding a Shape Function

In general, a good guess for the shape function is $w(z) = \lim_{n\to\infty} n\, w_{n,zn}$ (Roura 2001) and, indeed, this will work out for our weights. We start by considering the behavior for large n of the terms $\mathbb{P}(I_r = zn+\rho)$ for $r = 1, 2, 3$, where ρ does not depend on n. Assuming $zn + \rho \in \{0, \ldots, n\}$, we compute

$$\begin{aligned}
\mathbb{P}(I_r = zn+\rho) &= \binom{N}{zn+\rho} \frac{(t_r+1)^{\overline{zn+\rho}}\,(k-t_r)^{\overline{(1-z)n-\rho}}}{(k+1)^{\overline{N}}} \\
&= \frac{N!}{(zn+\rho)!((1-z)n-\rho)!} \frac{\dfrac{(zn+\rho+t_r)!}{t_r!}\,\dfrac{((1-z)n-\rho+k-t_r-1)!}{(k-t_r-1)!}}{\dfrac{(k+N)!}{k!}} \\
&= \underbrace{\frac{k!}{t_r!(k-t_r-1)!}}_{=1/\mathrm{B}(t_r+1,k-t_r)} \frac{(zn+\rho+t_r)^{\underline{t_r}}\,((1-z)n-\rho+k-t_r-1)^{\underline{k-t_r-1}}}{n^{\underline{k}}}, \qquad (26)
\end{aligned}$$

and since this is a *rational* function in n ,

$$\begin{aligned}
&= \frac{1}{\mathrm{B}(t_r+1,k-t_r)} \frac{(zn)^{t_r}\,((1-z)n)^{k-t_r-1}}{n^k} \cdot \left(1 + O(n^{-1})\right) \\
&= \underbrace{\frac{1}{\mathrm{B}(t_r+1,k-t_r)} z^{t_r}(1-z)^{k-t_r-1}}_{=:w_r(z)} \cdot \left(n^{-1} + O(n^{-2})\right), \qquad (n\to\infty). \qquad (27)
\end{aligned}$$

Thus $n\,\mathbb{P}(J_r = zn) = n\,\mathbb{P}(I_r = zn - t_r) \sim w_r(z)$, and our candidate for the shape function is

$$w(z) = \sum_{r=1}^{3} w_r(z) = \sum_{r=1}^{3} \frac{z^{t_r}(1-z)^{k-t_r-1}}{\mathrm{B}(t_r+1,k-t_r)} .$$

Note that $w_r(z)$ is the density function of a $\mathrm{Dir}(t_r+1, k-t_r)$ distributed random variable.

It remains to verify condition (24). We first note using (27) that

$$n\,w_{n,zn} = w(z) + O(n^{-1}) . \qquad (28)$$

Furthermore as $w(z)$ is a *polynomial* in z, its derivative exists and is finite in the compact interval $[0, 1]$, so its absolute value is bounded by a constant C_w. Thus $w : [0, 1] \to \mathbb{R}$ is *Lipschitz-continuous* with Lipschitz constant C_w:

$$\forall z, z' \in [0,1] \;:\; |w(z) - w(z')| \le C_w\,|z - z'| . \qquad (29)$$

For the integral from (24), we then have

$$\begin{aligned}
\sum_{j=0}^{n-1} \left| w_{n,j} - \int_{j/n}^{(j+1)/n} w(z)\,dz \right| &= \sum_{j=0}^{n-1} \left| \int_{j/n}^{(j+1)/n} n\,w_{n,j} - w(z)\,dz \right| \\
&\le \sum_{j=0}^{n-1} \frac{1}{n} \cdot \max_{z\in\left[\frac{j}{n},\frac{j+1}{n}\right]} \left| n\,w_{n,j} - w(z) \right| \\
&\underset{(28)}{=} \sum_{j=0}^{n-1} \frac{1}{n} \cdot \left[\max_{z\in\left[\frac{j}{n},\frac{j+1}{n}\right]} \left| w(j/n) - w(z) \right| + O(n^{-1}) \right] \\
&\le O(n^{-1}) + \max_{\substack{z,z'\in[0,1]:\\ |z-z'|\le 1/n}} |w(z) - w(z')| \\
&\underset{(29)}{\le} O(n^{-1}) + C_w \frac{1}{n} \\
&= O(n^{-1}),
\end{aligned}$$

which shows that our $w(z)$ is indeed a shape function of our recurrence (with $d = 1$).

D.2 Applying the CMT

With the shape function $w(z)$ we can apply Theorem D.1 with $\alpha = 1, \beta = 0$ and $K = a$. It turns out that case 2 of the CMT applies:

$$\begin{aligned} H &= 1 - \int_0^1 z\, w(z)\, dz \\ &= 1 - \sum_{r=1}^{3} \int_0^1 z\, w_r(z)\, dz \\ &= 1 - \sum_{r=1}^{3} \frac{1}{\mathrm{B}(t_r+1, k-t_r)} \mathrm{B}(t_r+2, k-t_r) \\ &\underset{(13)}{=} 1 - \sum_{r=1}^{3} \frac{t_r+1}{k+1} = 0\,. \end{aligned}$$

For this case, the leading-term coefficient of the solution is $t_n \ln(n)/\tilde{H} = n \ln(n)/\tilde{H}$ with

$$\begin{aligned} \tilde{H} &= -\int_0^1 z \ln(z)\, w(z)\, dz \\ &= \sum_{r=1}^{3} \frac{1}{\mathrm{B}(t_r+1, k-t_r)} \mathrm{B}_{\ln}(t_r+2, k-t_r) \\ &\underset{(17)}{=} \sum_{r=1}^{3} \frac{\mathrm{B}(t_r+2, k-t_r)(H_{k+1} - H_{t_r+1})}{\mathrm{B}(t_r+1, k-t_r)} \\ &= \sum_{r=1}^{3} \frac{t_r+1}{k+1} (H_{k+1} - H_{t_r+1})\,. \end{aligned}$$

So indeed, we find $\tilde{H} = \mathcal{H}$ as claimed in Theorem 6.2, concluding the proof for the leading term.

As argued above, the error bound is obtained by a second application of the CMT, where the toll function now is $K \cdot n^{1-\epsilon}$ for a K that gives an upper bound of the toll function: $\mathbb{E}[T_n] - an \le K n^{1-\epsilon}$ for large n. We thus apply Theorem D.1 with $\alpha = 1 - \epsilon, \beta = 0$ and K. We note that $f_c : \mathbb{R}_{\ge 1} \to \mathbb{R}$ with $f_c(z) = \Gamma(z)/\Gamma(z+c)$ is a strictly *decreasing* function in z for any positive fixed c and hence the beta function B is strictly decreasing in all its arguments by (13). With that, we compute

$$\begin{aligned} H &= 1 - \int_0^1 z^{1-\epsilon}\, w(z)\, dz \\ &= 1 - \sum_{r=1}^{3} \frac{\mathrm{B}(t_r+2-\epsilon, k-t_r)}{\mathrm{B}(t_r+1, k-t_r)} \\ &< 1 - \sum_{r=1}^{3} \frac{\mathrm{B}(t_r+2, k-t_r)}{\mathrm{B}(t_r+1, k-t_r)} = 0\,. \end{aligned}$$

Consequently, case 3 applies. We already know from above that the exponent that makes H become 0 is $\alpha = 1$, so the $F_n = \Theta(n)$. This means that a toll function that is bounded by $O(n^{1-\epsilon})$ for $\epsilon > 0$ contributes only to the linear term in overall costs of Quicksort, and this is independent of the pivot sampling parameter $\mathbf{t}$. Putting both results together yields Theorem 6.2.

Note that the above arguments actually *derive* — not only prove correctness of — the precise leading-term asymptotics of a quite involved recurrence equation. Compared with Hennequin's original proof via generating functions, it needed less mathematical theory.