Java 7's Dual Pivot Quicksort

Master Thesis

Supervised by
Prof. Dr. Markus E. Nebel
AG Algorithmen und Komplexität
Fachbereich Informatik
Postfach 3049, 67653 Kaiserslautern

Technische Universität
KAISERSLAUTERN

Sebastian Wild

February 18, 2013

Dedicated to my loving wife Lydia,
who had to put up with me during the creation of this thesis,
suffering from the little time I had for her and
from my bad mood when I spent too little time with her.

Abstract

Recently, a new Quicksort variant due to Yaroslavskiy was chosen as standard sorting method for Oracle's Java 7 runtime library. The decision for the change was based on empirical studies showing that on average, the new algorithm is faster than the formerly used classic Quicksort. Surprisingly, the improvement was achieved by using a dual pivot approach — an idea that was considered not promising by several theoretical studies in the past. In this thesis, I try to find the reason for this unexpected success.

My focus is on the precise and detailed average case analysis, aiming at the flavor of Knuth's series "The Art of Computer Programming". In particular, I go beyond abstract measures like counting key comparisons, and try to understand the efficiency of the algorithms at different levels of abstraction. Whenever possible, precise expected values are preferred to asymptotic approximations. This rigor ensures that (a) the sorting methods discussed here are actually usable in practice and (b) that the analysis results contribute to a sound comparison of the Quicksort variants.

Zusammenfassung

Vor einiger Zeit stellte Yaroslavskiy eine neue Quicksort-Variante vor. Diese wurde als Standardsortiermethode für Oracles Java 7 Laufzeitbibliothek ausgewählt. Die Entscheidung wurde aufgrund von empirischen Laufzeitstudien getroffen, die zeigten, dass der neue Algorithmus schneller ist als der bisher verwendete, klassische Quicksort. Überraschenderweise wurde diese Verbesserung durch den Einsatz von zwei Pivot-Elementen erreicht – eine Idee, die auf Basis theoretischer Untersuchungen in der Vergangenheit als wenig hilfreich angesehen wurde. In meiner Masterarbeit gehe ich u. a. diesem unerwarteten Erfolg auf den Grund.

Dabei betrachte ich schwerpunktmäßig präzise und detaillierte Average Case Analysen, nach dem Vorbild von Knuth s Reihe „The Art of Computer Programming". Insbesondere beschränke ich mich nicht darauf, abstrakte Kostenmaße wie die Anzahl von Elementaroperationen zu bestimmen, sondern versuche die Effizienz der Algorithmen auf verschiedenen Abstraktionsebenen zu verstehen. Diese Genauigkeit im Detail stellt sicher, dass (a) die hier vorgestellten Sortierverfahren in der Praxis verwendbar sind und (b) dass die durchgeführten Analysen zu einem fundierten Vergleich der Quicksort-Varianten herangezogen werden können.

> " *There's still one good thesis left in Quicksort.* "
>
> — D. E. Knuth quoted in the Acknowledgement of [Sed75]

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich die vorliegende Masterarbeit mit dem Thema „Java 7's Dual Pivot Quicksort“ selbstständig verfasst habe und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen, die anderen Werken dem Wortlaut oder dem Sinne nach entnommen wurden, habe ich durch die Angabe der Quelle, auch der benutzten Sekundärliteratur, als Entlehnung kenntlich gemacht.

(Ort, Datum)

(Unterschrift)

Contents

List of Algorithms

List of Listings

List of Tables

List of Figures

1 Introduction

Sorting is an elementary, yet non-trivial problem of considerable practical impact, and it is certainly among the most well-studied problems of computer science. As of this writing, Wikipedia lists 41 (!) different sorting methods, and at least a handful of those are known to every computer science student. Given this vast number, it is somewhat surprising to see that so many sorting methods used in practice are variants of one basic algorithm: Quicksort.

Due to its efficiency in the average, Quicksort has been used for decades as general purpose sorting method in many domains, e. g. in the C and Java standard libraries or as UNIX's system sort. Since its publication in the early 1960s by Hoare [Hoa62], classic Quicksort (Algorithm 1) has been intensively studied and many modifications were suggested to improve it even further, one of them being the following: Instead of partitioning the input file into two subfiles separated by a single pivot, we can create s partitions out of $s-1$ pivots.

Sedgewick considered the case $s = 3$ in his PhD thesis [Sed75]. He proposed and analyzed the implementation given in Algorithm 7. However, this dual pivot Quicksort variant turns out to be clearly inferior to the much simpler classic algorithm. Later, Hennequin studied the comparison costs for any constant s in his PhD thesis [Hen91] for his "reference Quicksort" given in Algorithm 5, but even for arbitrary $s \geqslant 3$, he found no improvements that would compensate for the much more complicated partitioning step. These negative results may have discouraged further research along these lines.

Recently however, in 2009, Yaroslavskiy proposed the new dual pivot Quicksort implementation as given in Algorithm 8 at the Java core library mailing list[1]. He initiated a discussion claiming his new algorithm to be superior to the runtime library's sorting method at that time: the widely used and carefully tuned variant of classic Quicksort from [BM93]. Indeed, Yaroslavskiy's Quicksort has been chosen as the new default sorting algorithm in Oracle's Java 7 runtime library after extensive empirical performance tests.

In light of the results on multi-pivot Quicksort mentioned above, this is quite surprising and asks for explanation. Accordingly, the average case performance of Algorithm 8 is studied in this thesis at different levels of abstraction. A preliminary version of this work mainly covering Chapter 4 appeared in [WN12].

[1] The discussion is archived at `http://permalink.gmane.org/gmane.comp.java.openjdk.core-libs.devel/2628`.

This thesis is organized as follows: In Chapter 2, some common definitions are made. Chapter 3 summarizes related work on Quicksort and its variants. It is remarkable to see that previous studies of dual pivot Quicksort came to the conclusion that two pivots are an unfavorable choice.

In Chapters 4 and 5 the main analytic work is done. There, I compute exact expected numbers of swaps and comparisons needed by the considered dual pivot Quicksort variants, including Yaroslavskiy's new algorithm. It turns out that Yaroslavskiy's partitioning method can take advantage of asymmetries in the outcome of comparisons to reduce their overall expected number. To the author's knowledge, this is the first explanation why this new dual pivot Quicksort might be superior to older variants. The same idea can also be used to significantly improve the dual pivot Quicksort variant proposed by Sedgewick in his PhD thesis [Sed75].

The expected values computed in Chapters 4 and 5 are evaluated against empirically determined numbers in Chapter 6. The variance of the numbers are empirically found to behave similar as for previously studied Quicksort variants.

In Chapter 7, I consider implementations of the algorithms on two particular machines. The first one is Knuth's mythical computer MMIX, which is used in "The Art of Computer Programming" for implementation and analysis of algorithms. The second machine is the Java Virtual Machine, where I count the expected number of executed Bytecode instructions. On both machines, Yaroslavskiy's dual pivot Quicksort is more efficient than the one by Sedgewick, even when incorporating the above mentioned improvement.

Complementing the theoretical analysis, Chapter 8 presents a runtime study of the algorithms on different runtime platforms.

Up to now, I only studied the very basic Quicksort algorithm, without the modifications surveyed in Chapter 3. In Chapter 9, I consider the arguably most successful of these variations in more detail: Selecting pivots as order statistics of a fixed size sample of the current list. Here, it turns out that asymmetric order statistics are superior for dual pivot Quicksort to choosing equidistant pivots. Finally, Chapter 10 summarizes the contributions of this thesis and outlines directions of future research.

> " *I have made this longer, because I have not had the time to make it shorter.* "
> —Blaise Pascal ("Lettres provinciales", letter 16, 1657)

> " *This report, by its very length, defends itself against the risk of being read.* "
> —Winston Churchill

> " *The tale grew in telling.* "
> —J. R. R. Tolkien (on the length of "Lord of the Rings")

2 Foundations

2.1 Common Notation & Identities

Here, I summarize some common mathematical notations used throughout this thesis. All of those should be fairly standard and probably the reader will find them familiar. Yet, I collect the definitions here to make the thesis self-contained.

In the course of the thesis, the reader will additionally encounter a few less common and self-made notions. Those will be defined on the spot with more explanatory comments than the brief general purpose conventions here. I try to repeat or link to definitions of those uncommon notations, whenever we encounter them.

- $[n..m] := \{n, \ldots, m\} \subset \mathbb{Z}$ for $n, m \in \mathbb{Z}$
- $[n] := [1..n]$
- $\mathcal{H}_n := \sum_{i=1}^{n} \frac{1}{i}$ is the nth harmonic number.
  Note that the harmonic numbers have the following asymptotic approximation:
  $$\mathcal{H}_n = \ln n + \gamma + \mathcal{O}(n^{-1}) \qquad \text{eq. (6.66) of [GKP94]} \qquad (\mathcal{H}_\infty)$$
- $[\mathit{condition}] := \begin{cases} 1 & \text{if } \mathit{condition} \text{ is true} \\ 0 & \text{otherwise} \end{cases}$
  This notation is called the Iverson-bracket and heavily used in [GKP94].
- For a random variable X, I denote by $\mathbb{E}\, X$ its expectation.
  Moreover, if Y is a second random variable such that $Y = f(X)$ for some (measurable) function f, then I write $\mathbb{E}_X\, Y$ for $\mathbb{E}\, Y$ if the dependency on X is to be stressed.
- One combinatorial identity keeps popping up in so many places that I found it more economically to cite it here once and for all. It appears as equation (6.70) in [GKP94, page 280] and says
  $$\sum_{0 \leqslant k < n} \tbinom{k}{m} \mathcal{H}_k = \tbinom{n}{m+1} \left(\mathcal{H}_n - \tfrac{1}{m+1} \right) \qquad \text{for integer } m \geqslant 0\ . \qquad (\Sigma i \mathcal{H}_i)$$
  Most of the time, I use it for $m = 1$, hence the nick name "$(\Sigma i \mathcal{H}_i)$".

2.2 Sorting: A Problem Definition

One reason why Quicksort is often used in practice is that it can be implemented *in-place*, i. e. if the input list is given as a random access array of memory, we can directly sort this list without having to copy the input. All Quicksort implementations studied in this thesis are of this kind. In fact, [Knu98, exercise 5.2.2–20] shows that Quicksort is guaranteed to get along with $\mathcal{O}(\log n)$ additional memory, if we apply tail-recursion elimination and avoid sorting the largest sublist first.

As a consequence, an in-place implementation of Quicksort can only work with a constant number of array elements at a time directly, i. e. without reading them from the array. Otherwise, more memory is required. Indeed, the implementations considered here will only read two array elements — and potentially write them at a different location — before loading the next elements. We will refer to this process of loading two elements and storing them again as one *swap*.

For the mathematically inclined reader, let us define the above terms properly. We are given n elements $A[1], \dots, A[n]$ in a data structure A, where the elements are taken from a totally ordered universe $A[i] \in \mathcal{U}$ for $i = 1, \dots, n$. A offers the operations of a random-access array:

- For index $i \in [n]$, we can **read** the value of the ith cell: $tmp := A[i]$.
- We can **write** the cell likewise: $A[i] := tmp$.

It is convenient to abbreviate by "Swap $A[i]$ and $A[j]$" the two (successive) reads and writes that exchange $A[i]$ and $A[j]$:

$$tmp := A[i] \qquad A[i] := A[j] \qquad A[j] := tmp\ .$$

Asking whether $A[i] < A[j]$ is called a **key comparison** (or just comparison for short).

The **in-place sorting problem** consists of rearranging A using the above operations such that

$$A[1] \leqslant A[2] \leqslant \cdots \leqslant A[n]$$

and the multiset of elements is the same as at the beginning, i. e. if $x \in \mathcal{U}$ occurs exactly $k \in \mathbb{N}$ times in the initial A, it occurs exactly k times in the sorted A.

Example 2.1: Let $\mathcal{U}$ contain pairs of strings $(name, tel)$, where *name* is a person's name and the *tel* is her telephone number. We order the pairs lexicographically. Then, the sorting problem amounts to preparing a (printed) telephone book from a list of raw (unordered) telephone number entries.

Assume, we are given the entries $A[1] = (\text{Mary}, 01143/7663)$, $A[2] = (\text{Adam}, 06543/2211)$, $A[3] = (\text{Zacharias}, 07654/3211)$ and $A[4] = (\text{Alice}, 0485/99887)$. Now, apply the following sequence of swaps: Swap $A[1]$ and $A[2]$, Swap $A[2]$ and $A[4]$ and Swap $A[3]$ and $A[4]$.

Then, we obtain the sorted telephone book: $A[1] = (\text{Adam}, 06543/2211)$, $A[2] = (\text{Alice}, 0485/99887)$, $A[3] = (\text{Mary}, 01143/7663)$ and $A[4] = (\text{Zacharias}, 07654/3211)$.

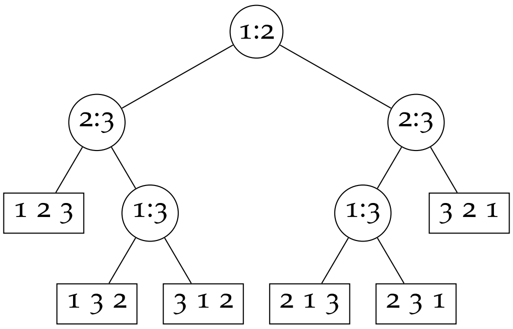


Figure 1: Example of a comparison tree for three elements. A label "1:2" in an inner node means compare the first and second element of the input. This example is taken from [Knu98, Fig. 34]

2.3 Complexity of Comparison Sorting

For sorting as defined in Section 2.2 in terms of swaps and comparisons, we can give general bounds on the number of these operations needed by *any* thinkable — or unthinkable — sorting method based on these abstract operations. The only restriction needed is that the method works correctly on *any* possible input list. But if this is not the case, the method does not deserve the term algorithm.

Comparisons The simplest of these bounds has become folklore under the term *information theoretic lower bound on comparison sorting*. It states that for any comparison based sorting algorithm and any input size n, there exists an input list such that the algorithm needs at least

$$\lceil \log_2(n!) \rceil = n \log_2 n - \tfrac{1}{\ln 2} n + \tfrac{1}{2} \log_2 n + \mathcal{O}(1)$$

comparisons to sort this list. Accordingly, comparison based sorting has linearithmic worst-case complexity.

The proof is described so nicely in [Knu98, Section 3.5.1] that I only recite the basic idea. From an information theoretic standpoint, sorting a list means identifying the permutation that is needed to transform it into sorted order. Any (deterministic and sequential) comparison based sorting algorithm then induces a *comparison tree*: At some point, the algorithm does its first comparison, which becomes the root of the comparison tree. Depending on the outcome of this comparison, we continue in the left respectively right subtree of the root. So, when the next comparison is encountered it is attached as left respectively right child. This process continues, until we know the input permutation, which then gets added as leaf node. Each single input gives one path; all inputs combined form the comparison tree. Determinism of the algorithm ensures that the collection of these paths actually forms a tree. There will be a leaf for every possible permutation, of which there are $n!$ in total. An example for a comparison tree is shown in Figure 1.

A worst case input w. r. t. the number of comparisons now corresponds to a longest path from the root to a leaf. As the comparison tree is a binary tree with $n!$ leaves, we can fit at most 2^k nodes on level k. Therefore, we have at least one path with length at least $\lceil \log_2(n!) \rceil$ in every comparison tree.

Comparison trees can also be used to derive a lower bound on the *average* number of comparisons needed for sorting. It is a trivial observation that the *external path length* in the comparison tree equals the cost of sorting every of the $n!$ possible input permutations once. So dividing it by $n!$ yields the average number of comparisons. It can be shown that the external path length of a binary tree is minimal iff all its leaves are located on at most two consecutive levels [Knu98, exercise 5.3.1-20]. Together with the knowledge from above that at least one path has length $\lceil \log_2(n!) \rceil$, we find that any comparison based sorting algorithm needs at least

$$\log_2(n!) + \mathcal{O}(1) = n \log_2 n - \tfrac{1}{\ln 2} n + \mathcal{O}(\log n)$$

comparisons on average to sort a random list of length n. Note that this matches the lower bound for the worst case costs up to minor terms. In fact, in a sorting algorithm with the above described optimal comparison tree, best case and worst case differ by *exactly one comparison*, independent of n!

One might ask whether more involved arguments might improve upon these bounds. Hardly. Algorithms are known which show that the bounds are *asymptotically tight*: Samplesort needs $n \log_2 n + \mathcal{O}(n)$ comparisons in expectation and binary insertion sort even achieves this bound in the worst case. For the worst case bound, KNUTH further describes merge insertion sorting, which has been proven to achieve the information theoretic lower bound *exactly* for $n \in \{1, \ldots, 12, 20, 21\}$. So, we might assume this bound rather tight.

Swaps For any list, $n - 1$ exchanges suffice to sort the list, if we are allowed to do an arbitrary number of comparisons in between. This is easily seen as selection sort works that way: In each step, we use comparisons to select the minimum of the not yet sorted range. Then one exchange brings this element to its final position. In the very last step, there is only one other element left, so the last exchange always puts two elements into their final positions.

One can give a simple lower bound on the number of write operations in the array. Each element that is not already at its final position has at least to be written once, namely at the place it belongs. There are plenty of permutations without any fix point, e. g. $2\,3 \ldots n\,1$, so n write operations are needed in the worst case.

Towards a lower bound for the average case, we determine the expected number of fix points in a random permutation. To this end, let X_i be the indicator variable for the event "i is a fix point". As there are exactly $(n-1)!$ permutations of $[n]$ that leave index i as fix point, we have $\mathbb{E}\, X_i = \frac{(n-1)!}{n!} = \frac{1}{n}$. Now, the expected number of fix points of the total permutation is $\mathbb{E} \sum_{i=1}^{n} X_i = \sum_{i=1}^{n} \mathbb{E}\, X_i = 1$. Therefore, any sorting algorithm needs at least $n - 1$ write operations in the average.

Whereas we could give rather useful lower bounds on the number of comparisons required by any sorting algorithm, the lower bounds for swaps appear rather weak — even though they are tight: Simple methods achieve the bounds[2] — at the price of an excessive number of comparisons. The high art of sorting is to find algorithms with a balance between the number of needed comparisons and swaps. As we will see, Quicksort can be counted among those.

2.4 Why Analyze Algorithms?

Analysis of algorithms can be seen to serve two independent goals. First of all, it tries to let us compare different algorithms for the same problem. This is an application-driven point of view: Given a problem, *which* is the best-suited algorithm to solve the problem.

The second goal of algorithm analysis is more academic in motivation — yet not less relevant to applications, as algorithmic improvements often arise from it: *Why* is a given algorithm well-suited for a problem and another is not.

Assuming time is the scarcest resource, the ultimate goal for comparing algorithms is to predict the *actual time* (in seconds) needed for a program to terminate when run on a particular machine and on a particular input. After all, it is this runtime that we would like to minimize in an application. This goal is hardly achievable even for the simplest of programs.

As the effort a program undertakes can vary very much from input to input, a typical attempt to simplify the problem is by considering *average behavior*: Given some distribution of inputs, we try to determine the expected runtime of the program for random inputs drawn according to the given distribution. In this global form, i. e. computing a single expected value over all possible inputs, the analysis will not be fruitful. Algorithms have to deal with infinitely many different input instances, so these instances will in some sense grow beyond any bound. It is rather natural to assume that the runtime of non-trivial algorithms will then grow beyond any bound, as well, when they process larger and larger instances.

Let us fix a notion of *size* of inputs, i. e. each input is assigned a non-negative integer such that for any size n, there are only finitely many inputs of this size. Note that there are many ways to define size, even though natural definitions are at hand most of the time. For sorting algorithms, we define the size of an instance to be the length of the list to sort, i. e. the number of elements in the list. Now that we have agreed upon a size for every input, we can determine *conditional expected runtimes* of a program, given that the input has size n. For a sensible size measure, we expect this conditional expected runtime to somehow increase as n gets large, even though examples show that non-monotonic behavior can occur at a fine-grained level.

[2] For n write operations in the worst case, we have to tweak selection sort further: We always take one element x "in our hand" and find its rank r by counting the number of smaller elements in the whole list. Then, we put x into $A[r]$ and continue with the element found there. With some caution we can make sure that the whole list is sorted in the end, and each misplaced element is written exactly once in the array.

One might question whether these average values provide enough detail to still yield suitable runtime predictions. Whereas inputs can indeed differ considerably from the average, for many problems the variance of runtimes among different inputs of the same size is reasonably small. And of course, by the law of large numbers, the mean runtime of many executions of the algorithm on random inputs converges to the expected runtime.

So, we relaxed our ultimate goal a little by only asking for expected runtimes for a given input size n, on a particular machine. Still, these expected runtimes will vary *from machine to machine*; even more: As different machines have different machine languages, strictly speaking, we cannot run the same program on different machines. The best we can do is use 'similar' programs.

It is time to make our subject of study a little more precise: In this thesis, a **program** is a description of behavior that can be executed on a particular machine. Typically, programs are built with some more abstract idea in mind. We will refer to such an idea as **algorithm**. Several different programs — e. g. for different machines — can *implement* the same algorithm. Sometimes, one might also say that a more abstract algorithm is implemented by some less abstract algorithm, which fixes some aspects of realization, but is not yet a full program.[3]

Admittedly, this abstraction concept is very vague. Its only purpose here is to make clear: By considering *runtime*, we can only compare *programs*, not algorithms, as the latter are not directly executable and hence, runtime is not defined for those. This is an unsatisfactory situation, as a slightly different implementation of the same algorithm might have a quite different runtime.

I would like to close this line of thought with the statement that it is in general quite hard to compare algorithms if actual runtime is our concern — or stated differently: The study of runtime is limited in the insight it provides into algorithms. All properties of the algorithm are reduced to one number which is influenced by so many aspects — e. g. the runtime of different processor instructions, memory hierarchies, interrupts by the operating system etc. — that the resulting system appears chaotic.

If we consider analysis of algorithms as a means to understand *why* certain algorithms behave differently, our point of view changes. For that, we would like to model algorithms *as abstractly as possible, but as concretely as needed* as to still observe the behavior that separates the algorithms.

A popular model that allows to study abstract algorithms is the *elementary operation model*. For many problems, we can identify a set of operations with two properties:

(1) All algorithms for the problem use these operations, i. e. by a certain sequence of these elementary operations, we can solve the problem. This ensures that algorithms essentially differ in the operation sequences they imply.

[3] This idea of abstraction/refinement relations between algorithms or programs can be made formal, if we fix a semantic for abstract algorithms. One such approach are *abstract state machines*, see e. g. [BS03]. We will confine ourselves to the intuitive understanding.

(2) The runtime of programs is highly (and positively) correlated with the number of executed elementary operations. As long as every innermost loop contains at least one elementary operation, this will hold.

For comparison-based sorting, we have the elementary operations *key comparison* and *swap*, see Section 2.2.

If we now determine the expected number of elementary operations that an algorithm executes for random inputs of size n, we can get *insight* into the performance of the algorithm. Especially, we can determine the order of growth of the number of operations used as n increases.

Sometimes, the elementary operation model is too coarse to capture a desired effect. We can make our cost model more concrete by counting *primitive instructions*. In this model, we incorporate the cost contribution of *every* instruction needed to realize an algorithm, whereas the elementary operations model typically ignores many instructions such as jumps and branches for program logic. Of course, this model requires a much more detailed description of the algorithm than the elementary operation model.

By *understanding* the underlying mechanisms which and how many instructions different algorithms need, we can develop expert knowledge about algorithms. This knowledge can give us an "informed feeling", *which* algorithm might be best suited for a given problem at hand. Thereby, even if we are originally only interested in the question, *which* algorithm to use, one should try to approach the question *why* algorithms behave differently. Chances are good that this helps comparing algorithms, as well.

2.5 Knuthian Analysis of Algorithms

In his famous book series "The Art of Computer Programming", Knuth popularizes a certain methodology for the average case analysis of algorithms, which I refer to as **Knuthian analysis**. It allows to determine the *expected costs* of an algorithm under a given *additive* cost model and input distribution. Let us first properly define the occurring terms. Assume, we label the program's instructions with "line numbers" $\ell_1, \ldots, \ell_k$. Then, each executed instruction in a *terminating* run of the program bears such a label ℓ_i, namely its line number in the code. I call the sequence of visited line numbers the **trace** of this execution. Formally, a trace t is a word over $\Sigma := \{\ell_1, \ldots, \ell_k\}$, so I write $t \in \Sigma^\star$.[4]

These traces are now used to define **additive cost measures**. A cost measure c is characterized by a cost contribution $c(\ell_i) \in \{0, 1, 2, \ldots\}$ for each line number $\ell_i \in \Sigma$. So, the cost contribution of one instruction depends only on the line number in the program listing, but not on any context information from the trace. Then, the cost $c(t)$ of a trace $t = t_1 \ldots t_m$ is defined by *summation* over its elements' costs, i. e. $c(t) := c(t_1) + \cdots + c(t_m)$. (Hence the name *additive* cost model.)

[4] According to my definition, a non-terminating execution does not have a well-defined trace. Of course, the definition of traces can be generalized to include infinite words, which then naturally imply infinite cost. As it complicates notation, but does not help us for the analysis of algorithms, I simply assume that non-terminating runs do not occur for the considered inputs.

Now, we can properly define expected costs: Let I be a random input drawn according to the given input distribution and $t(I)$ be the trace of the program when processing I. Then the **expected cost $\mathbb{E}\, c$ of a program w. r. t. a given input distribution** is defined as

$$\mathbb{E}\, c := \mathbb{E}_I\, c\bigl(t(I)\bigr) \; .$$

A simple example of an additive cost model is the number of executed instructions. This model simply assigns cost 1 to every line number ℓ_i in a program. Another example is the number of memory references, which assigns 1 to instructions that access the main memory and 0 to all others.

The essential idea of Knuthian analysis is to compute the expected **frequency** f_i of every instruction ℓ_i in the program, i. e. how often the corresponding line is reached in expectation when the program is executed on a random input, drawn according to the given input distribution. Then, we multiply the *cost contribution of each line* by the corresponding frequency and *sum over all lines*.

In general, both the frequencies *and* the cost contributions of line numbers can be random variables. Then, Knuthian analysis still yields the correct expected value if the cost contributions are stochastically independent of the input. The correctness is easily checked by computing:

$$\begin{aligned}
\mathbb{E}_I\, c\bigl(t(I)\bigr) &= \sum_{\text{input } I} \Pr(I) \cdot \mathbb{E}\, c\bigl(t(I)\bigr) \\
&= \sum_{\text{input } I} \Pr(I) \cdot \sum_{i=1}^{k} \bigl|t(I)\bigr|_{\ell_i}\, \mathbb{E}\, c(\ell_i) \\
&= \sum_{i=1}^{k} \mathbb{E}\, c(\ell_i) \cdot \underbrace{\sum_{\text{input } I} \Pr(I) \cdot \bigl|t(I)\bigr|_{\ell_i}}_{} \\
&= \sum_{i=1}^{k} \mathbb{E}\, c(\ell_i) \cdot \qquad \mathbb{E}_I\, f_i \; .
\end{aligned}$$

The most important special case where the cost contributions $c(\ell_i)$ are trivially independent of the input are **constant additive cost models**. Here, the cost contributions are simply given by a function $c : \{\ell_1, \ldots, \ell_k\} \to \mathbb{N}_0$. Both examples from above — the number of executed instructions and the number of memory references — are in fact constant models, so Knuthian analysis can be applied to them.

As indicated in Section 2.4, we have to compute the expected costs of an algorithm w. r. t. some *input size* measure $n = n(I)$. Therefore, we actually take a *family* $\mathcal{P} = \{P_n\}_{n \in \mathbb{N}}$ of input distributions, such that each P_n only contains inputs of size n. Then, we determine the frequencies f_i as *functions* in n, i. e. $\mathbb{E}\, f_i = \mathbb{E}\, f_i(n) = \mathbb{E}_{P_n} f_i$ and

$$\mathbb{E}\, c(n) = \sum_{i=1}^{k} c(\ell_i) \cdot \mathbb{E}_I\, f_i(n) \; .$$

2.5.1 Language Theoretic Interpretation of Knuthian Analysis

The restriction to additive cost measures is a serious limitation, as discussed in the following section. The trace of an execution can give us much more information. However the trace language of a program can be a rather nasty fellow. It is related to the language of *valid computations* of a TURING machine, which is known to be not context-free in general. For the trace language, it might be suspected that even context-sensitive descriptions are too weak, as the valuations of variables are not directly encoded in the trace.

Therefore, one tries to transform the trace language into an entity that is easier to handle. The restriction to *additive* cost models corresponds to taking the *commutative image* (a. k. a. PARIKH-image) of the trace language. These commutative languages are much easier, for example PARIKH's famous theorem [Par66, Corollary 1] says that the commutative image of a context-free language is always regular.

For the average case analysis, we have to incorporate the probability *distribution* of inputs. It naturally induces a distribution over traces via

$$\Pr[t] = \sum_{\substack{\text{input } I:\\ t(I)=t}} \Pr[I] \;.$$

This gives us a language $L \in \Sigma^\star$ with a probability distribution on the words $w \in L$, which can be fully encoded as *formal power series* $G(\ell_1, \dots, \ell_k)$ over the non-commutative monoid $(\Sigma, \cdot)$ with real coefficients [CS63]:

$$G(\ell_1, \dots, \ell_k) = \sum_{w \in L} \Pr[w] \cdot w \;.$$

The transition to the commutative image $\tilde{L}$ of L now translates to making $(\Sigma, \cdot)$, i. e. the variables $\ell_1, \dots, \ell_k$ *commutative*:

$$\tilde{G}(\ell_1, \dots, \ell_k) = \sum_{w \in L} \Pr[w] \cdot \ell_1^{|w|_{\ell_1}} \cdots \ell_k^{|w|_{\ell_k}} \;,$$

where $|w|_{\ell_i}$ denotes the number of ℓ_is in w. Finally, we are not interested in the commutative images of traces themselves, but on their costs in the additive constant cost model c. Therefore, we apply the homomorphism h_c from $(\Sigma, \cdot)$ to the commutative monoid of monomials $Z = \big(\{z^i : i \in \mathbb{N}_0\}, \cdot\big)$ defined by

$$h_c(\ell_i) := z^{c(\ell_i)} \;.$$

As Z is commutative, $h(\tilde{L}) = h(L)$ holds. The formal power series of $h(L)$ is then

$$G_c(z) = \sum_{w \in L} \Pr[w] \cdot z^{c(w)} = \sum_{w \in L} \Pr[w] \cdot \big(z^{c(\ell_1)}\big)^{|w|_{\ell_1}} \cdots \big(z^{c(\ell_k)}\big)^{|w|_{\ell_k}} \;,$$

which is the probability generating function for the cost of a random $w \in L$. The expectation can then be computed as $\mathbb{E}\, c = G_c'(1) = \sum_{w \in L} \Pr[w] \cdot c(w)$.

2.5.2 Limitations of Knuthian Analysis

There are two main limitations of KNUTHian analysis:

(1) *Additive cost models.*
Not all interesting cost models are additive. For example the maximal amount of memory required is not additive as we can allocate and free memory alternatingly. (The total amount of memory allocated during the execution is additive, which bounds the space complexity from above. However, this bound can be arbitrarily bad.)
When it comes to actual *running time*, additive cost models are an idealized view of modern computers. The time taken for a single instruction can depend on the state of the instruction pipeline, the contents of the caches, the size of the operands, and so on. Thus, actual running time is not an additive cost model, either.

(2) *Distribution of costs.*
The correctness argument above relies on the linearity of the expected value. In general, it is not possible to determine more information on the distribution of costs by KNUTHian analysis even for simple cost models — unless all frequencies happen to be stochastically independent. Note in particular that it is not sufficient to determine the variance of individual frequencies and add them up to get the variance of the costs.

Despite these limitations, KNUTHian analysis provides valuable insights into the inner workings of an algorithm.

> “ *As far as the laws of mathematics refer to reality, they are not certain; and as far as they are certain, they do not refer to reality.* ”
>
> — ALBERT EINSTEIN in "Geometry and Experience", Jan 27, 1921

3 Classic Quicksort and its Variants: Previous Work

> “ *If I have seen further, it is by standing on the shoulders of giants.* ”
>
> — Isaac Newton

3.1 Abstract Description of Quicksort

Abstractly stated, Quicksort is a straight-forward application of the divide-and-conquer paradigm to comparison based sorting:

Given a list A of n elements, choose one element p as the *pivot*. Rearrange the list such that all elements left of p are smaller than p, and all elements right of p are larger than p. Obviously then, p has found its final position in the sorted list. Applying this procedure recursively to the parts left and right of p completes sorting of the whole list.

An immediate and natural generalization of this algorithm is given by the following idea: We divide the list into $s \geqslant 2$ sublists instead of just two. To this end, we choose $s-1$ pivot elements $p_1 \leqslant \cdots \leqslant p_{s-1}$ and partition the list such that in the ith sublist, all values lie between p_{i-1} and p_i for $i = 1, \ldots, s$, where we set $p_0 = -\infty$ and $p_s = +\infty$.

3.2 Classic Quicksort

A priori, there is not *the* prototypical implementation of the abstract Quicksort idea. However, there is a pattern in the implementation of in-place Quicksort that deserves the predicate ‘classic’: Hoare’s crossing pointers technique. It is present in the very first publication of Quicksort [Hoa61a] and is very nicely described in [Hoa62]. Two pointers i and j scan the array from left and right until they hit an element that does not belong in their corresponding subfile. More specifically, first i moves right until an element greater than the pivot is found. This element obviously belongs in the upper part. Then, j moves left until an element smaller than the pivot is found — this element likewise belongs in the lower part. Then the elements $A[i]$ and $A[j]$ are exchanged, such that afterwards both elements are in the correct part of the list. Then, scanning continues. Once the two pointers have crossed, we have found the boundary between small and large elements. Accordingly, we can put the pivot element at its final position, finishing the partitioning step.

The remarkable feature of this method is that by one swap operation, we move *two* elements to their final positions in the current partitioning step. This implies a trivial upper

Algorithm 1. Classic Quicksort implementation by SEDGEWICK as given and discussed in detail in [Sed75, Sed78]. We take the rightmost element as pivot instead of the leftmost, as it is done in Program 1.2 of [SF96].
Partitioning is done as follows: Two pointers i and j scan the array from left and right until they hit an element that does not belong in this subfile. Then the elements $A[i]$ and $A[j]$ are exchanged. This crossing pointers technique dates back to HOARE's original formulation of Quicksort [Hoa61a].

```
QUICKSORT($A, left, right$)
    // Sort the array $A$ in index range $left, \dots, right$.
    // We assume a sentinel value $A[left-1] = -\infty$, i. e. $\forall i \in \{left, \dots, right\} : A[left-1] \leqslant A[i]$
    if $right - left \geqslant 1$
        $p := A[right]$        // Choose rightmost element as pivot
        $i := left - 1;\ \ j := right$
        do
            do $i := i + 1$ while $A[i] < p$ end while
            do $j := j - 1$ while $A[j] > p$ end while
            if $j > i$ then Swap $A[i]$ and $A[j]$ end if
        while $j > i$
        Swap $A[i]$ and $A[right]$ // Move pivot to final position
        QUICKSORT($A$, $left$ , $i - 1$)
        QUICKSORT($A, i + 1, right$)
    end if
```

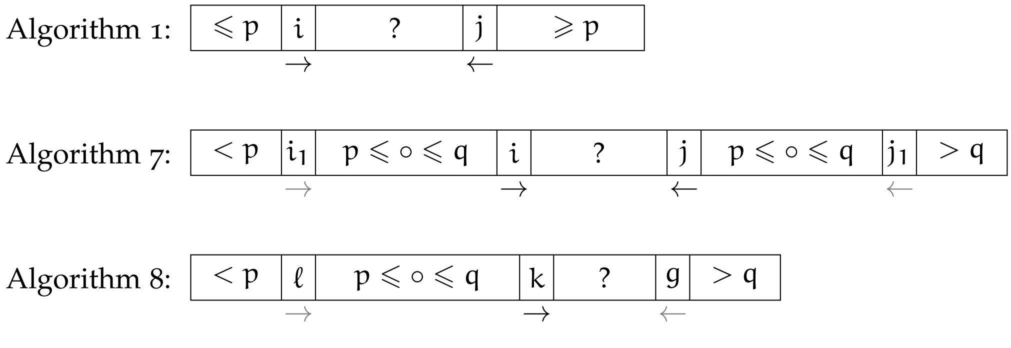


Figure 2: Comparison of the three different partitioning schemes used by the analyzed Quicksort variants. The pictures show the invariant maintained during partitioning.
Black arrows indicate the movement of pointers in the main loop of partitioning. Gray arrows show pointers that are moved only on demand, i. e. when an element is swapped to the range whose boundary the pointer defines.

bound of $\frac{1}{2}n$ swaps in the first partitioning step. Most studied variants and most practically used implementations of Quicksort use HOARE's crossing pointers technique.

Algorithm 1 shows a pseudocode implementation of abstract Quicksort that uses HOARE's crossing pointers technique. I will therefore call this algorithm **classic Quicksort**. As a real classic, Algorithm 1 is reduced to the essential and shines with elegant brevity. Despite its innocent appearance, classic Quicksort ranks among the most efficient sorting methods — precisely because its inner loops are so simple and short.

In the light of this, we may forgive it the idiosyncrasy of requiring a sentinel value in $A[0]$ that is less or equal to all occurring elements. Relying on such a sentinel value allows to omit range checks in the inner loops, and it thereby contributes significantly to efficiency and beauty of the algorithm. Both should be qualities of a classic and hence I adopted this sentinel trick.

Apart from that I tried to keep Algorithm 1 as 'basic' as possible. The history of Quicksort has been full of eager suggestions for clever improvements over the basic algorithm — but it has also seen many seemingly helpful variations being unmasked as mere drags for a classic Quicksort. Section 3.4 will tell part of this story.

3.3 Analysis of Classic Quicksort

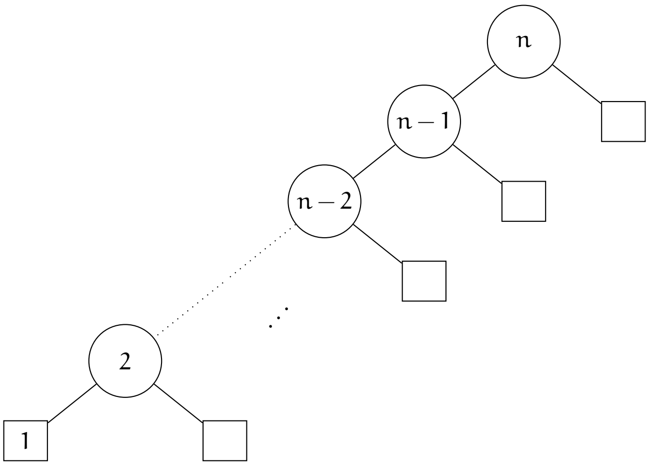
3.3.1 Worst Case

As Quicksort is comparison based, the lower bounds of 2.3 apply. However, a closer look at the algorithm shows that they are far from being tight here.

Consider the already sorted list $1\,2\ldots n$ and the behavior of Algorithm 1 on it. In the first partitioning step, we select the pivot $p = n$ and the first inner loop only terminates when $i = n$, as all other elements in the list are $< p$. So the inner loop is left with $i = n$. The swap after the loop exchanges p with itself and the recursive calls happen on ranges $[1..n-1]$ and $[n+1, n] = \varnothing$. The second call immediately terminates, but the first call finds itself in the same situation as an initial call for the sorted list $1\,2\ldots(n-1)$ of $n-1$ elements. So the same steps apply again. For $n = 1$, the algorithm terminates. Summing up the number of comparisons from all partitioning steps yields

$$(n+1) + (n) + (n-1) + \cdots + 4 + 3 = \tfrac{1}{2}n^2 + \tfrac{3}{2}n - 2\,.$$

With respect to the number of comparisons, this is actually the worst case for Algorithm 1: Every partitioning step of a sublist of length k needs exactly $k+1$ comparisons, irrespective of the pivot. Therefore, the number of comparisons only depends on the shape of the recursion tree. The tree corresponding to above input is a linear list disguised as degenerate tree:



With a little more scrutiny, one can prove that no recursion tree implies more comparisons than such a linear list.

It is interesting to note, though, that for sorting the above list, Algorithm 1 does not do any real swap — the swap in line 7 is never executed! We do have the swaps from line 9, but those are executed for any input list. So, this worst case for comparisons is actually rather good w. r. t. swaps.[5]

This poses the question, how the real worst case of overall costs looks like. For the abstract formulation of Algorithm 1, this overall costs are not well-defined. However, Sedgewick studies the worst case of a `MIX` implementation of his classic Quicksort[6] in [Sed75, Chapter 4]. There, he could show that in fact linear list type recursion trees are worst case instances for classic Quicksort w. r. t. overall runtime, even though they induce very few swaps.

3.3.2 Average Case — Elementary Operations

As we have seen in the last section, the worst case complexity of Quicksort is quadratic, which does not at all justify the name *Quick*sort. However, this worst case turns out to be very rare, such that on average, Quicksort is indeed quick. To quantify this, we assume a input distribution, namely the random permutation model, see Section 4.1.2. Under this model, we determine the expected number of swaps and comparisons Algorithm 1 needs to sort a random permutation of length n. The analysis is quite easy and well-known. It appears in more or less the same way in numerous sources, e. g. [SF96, Theorem 1.3], [Sed75, Chapter 3], [Sed77b], [Knu98, Section 5.2.2] and to some extend also in [CLRS09, Section 7.4]. Nevertheless will I briefly go through it once more, as the Quicksort variants the cited analyses are based upon differ in some details.

[5] Note that the exact number of swaps from line 9 equals the number of partitioning steps. The sorted list is actually a worst case instance w. r. t. the number of partitioning steps, therefore it is not best case w. r. t. swaps. It still is a 'good' case, though, as the number of swaps is within a factor of two of the best case.

[6] Sedgewick considers Program 2.4 from [Sed75], which is essentially Algorithm 1 but where the pivot is chosen as the first element of the list. Program 2.4 also includes special treatment of short sublists by Insertionsort, see Section 3.4.3.

The first crucial observation is that when Algorithm 1 is invoked on a random permutation of the set $[n]$, the ranges for the recursive calls again contain a random permutations of the respective elements $[1..p-1]$ and $[p+1..n]$. This allows to set up a direct *recurrence relation* for the expected costs. The argument is detailed in Section 4.2. The recurrence for the expected cost C_n of sorting a random permutation of length n sums over all possible values of the pivot p:

$$\begin{aligned} C_n &= pc_n + \tfrac{1}{n}\sum_{p=1}^{n}\bigl(C_{p-1}+C_{n-p}\bigr) = pc_n + \tfrac{2}{n}\sum_{p=1}^{n} C_{p-1} \qquad (n \geqslant 2) \\ C_0 &= C_1 = 0 \end{aligned}$$

where pc_n are the expected costs for the first partitioning step. For $n \leqslant 1$, Algorithm 1 immediately returns. The second equation for C_n follows by the symmetry of the sum. If we now consider the difference

$$\begin{aligned} & nC_n - (n-1)C_{n-1} = npc_n - (n-1)pc_{n-1} + 2C_{n-1} \qquad (n \geqslant 3) \\ \Longleftrightarrow \quad & nC_n - (n+1)C_{n-1} = npc_n - (n-1)pc_{n-1} \end{aligned}$$

Dividing by $n(n+1)$ yields a telescoping recurrence for $C_n/(n+1)$, which is trivially written as "closed" sum:

$$\begin{aligned} \frac{C_n}{n+1} &= \frac{C_{n-1}}{n} + \frac{npc_n - (n-1)pc_{n-1}}{n(n+1)} \\ &= \frac{C_2}{3} + \sum_{i=3}^{n} \tfrac{ipc_i - (i-1)pc_{i-1}}{i(i+1)} \ . \end{aligned} \tag{3.1}$$

The next step is to determine the expected number of swaps and comparisons in the first partitioning step. For the number of comparisons, we observe that whenever we compare two elements, we also move i or j one step up respectively down. For distinct elements, we always end with $i = j + 1$ after the loop. Thus in total, $n + 1$ pointer movements happen which imply $n + 1$ comparisons per partitioning step. Inserting $pc_n = n + 1$ in eq. (3.1) yields

$$C_n = (n+1)\Bigl(\tfrac{3}{3} + \sum_{i=3}^{n} \tfrac{i(i+1)-(i-1)i}{i(i+1)}\Bigr) = 2(n+1)(\mathcal{H}_{n+1} - \mathcal{H}_3 + \tfrac{1}{2}) \qquad (n \geqslant 3)\ .$$

The number of comparisons per partitioning step is independent of the chosen pivot. For swaps, the situation is slightly more complicated. The swap in line 7 of Algorithm 1 is executed for every pair of a large element in $A[1..p-1]$ and a small element in $A[p..n-1]$. For a given pivot p, the number of large elements in $A[1..p-1]$ is hypergeometrically distributed: We draw without replacement the positions of the $n-p$ large elements among the $n-1$ available positions and any of the $p-1$ first cells is a 'success'.[7] Therefore, the

[7] Note that this already determines the $p-1$ positions for the small elements.

	exact expectation	asymptotics (error $\mathcal{O}(\log n)$)
Comparisons	$2(n+1)(\mathcal{H}_{n+1} - \frac{4}{3})$	$2n \ln n - 1.5122n$
Swaps[a]	$\frac{1}{3}(n+1)(\mathcal{H}_{n+1} - \frac{1}{3}) - \frac{1}{2}$	$0.\overline{3}n \ln n + 0.0813n$

[a] This includes the swaps incurred at line 9 of Algorithm 1. For KNUTHian analysis, these have to be treated separately.

Table 1: Expected number of comparisons and swaps used by Algorithm 1 for sorting a random permutation of length n. The asymptotic approximation for $n \to \infty$ uses the expansion of harmonic numbers eq. ($\mathcal{H}_\infty$) on page 15.

expected number of such elements is the mean of this distribution, namely $(n-p)\frac{p-1}{n-1}$. The total expected number of swaps is then

$$\frac{1}{n}\sum_{p=1}^{n} \frac{(n-p)(p-1)}{n-1} = \frac{n-2}{6} .$$

Adding 1 for the swap in line 9 yields $pc_n = \frac{n+4}{6}$ for the number of swaps.[8] Using eq. (3.1) and the partial fraction decomposition of the summand gives

$$\begin{aligned} C_n &= (n+1)\Big(\tfrac{1}{3} + \sum_{i=3}^{n}\big(\tfrac{1}{2i} - \tfrac{1}{6(i+1)}\big)\Big) \\ &= (n+1)\big(\tfrac{1}{2}(\mathcal{H}_n - \mathcal{H}_2) - \tfrac{1}{6}(\mathcal{H}_{n+1} - \mathcal{H}_3) + \tfrac{1}{3}\big) \\ &= \tfrac{1}{3}(n+1)(\mathcal{H}_{n+1} - \tfrac{1}{3}) - \tfrac{1}{2} . \end{aligned}$$

3.3.3 Variances

Computing exact expected costs is pointless from a practical point of view, unless the actual costs are likely to be close to the mean of their distribution. The arguably easiest way to check that is to compute the *standard deviation* of the costs. Then, CHEBYSHEV's inequality states that a $1 - \frac{1}{k^2}$ fraction of all inputs cause costs of $\mu \pm k \cdot \sigma$ where μ are the expected costs and σ is the standard deviation. If further $\sigma \in o(\mu)$ for $n \to \infty$, this means that the relative deviation from the mean $\frac{\mu \pm k \cdot \sigma}{\mu} \to 1$ for any constant k; differently stated: For any constant probability p and error ϵ, there is a n_0 such that for $n \geqslant n_0$, the probability of a relative deviation from the mean of more than ϵ is less than p.

Bivariate generating functions provide a general approach to compute the variance and thus the standard deviation, see [SF96, §3.12]. There, the authors also derive the variance of the number of comparisons of classic Quicksort [SF96, Theorem 3.12]:

$$\sigma_{\text{cmps}} = \sqrt{7 - \tfrac{2}{3}\pi^2} \cdot n + o(n) .$$

[8] Note that this swap is often not included, as it occurs exactly once per partitioning step and thus can easily be treated separately. This is done e. g. in [Sed77b]. As I want to count elementary operations here without doing a KNUTHian analysis, I include the swap right away.

This result is originally derived by Knuth in [Knu98, exercise 6.2.2-8] via the correspondence between Quicksort and search trees described in Section 3.5.1.

The standard deviation of the number of swaps is much more difficult to handle, since their number in the first partitioning step and the sizes for recursive calls are stochastically dependent. Yet, Hennequin could derive in [Hen91, Proposition IV.7] asymptotics for the variance of the number of swaps used by classic Quicksort.[9] The exact term is quite lengthy, so I only give the result with rounded constants here. Thereby the standard deviation is approximately

$$\sigma_{\text{swaps}} \approx 0.02372n + o(n) \ .$$

Since the standard deviation for the number of both swaps and comparisons is linear in n, we indeed have the centralization property described above. Consequently, the expected values are good estimates of the actual costs for large n and computing exact expected costs is worth the effort.

3.3.4 Average Case — Processor Cycle Counts

Using the analysis of Section 3.3.2, one can easily compute the number of primitive instructions used by an implementation of Algorithm 1. I will do that in detail for two implementations in Chapter 7, where the details are discussed. Results are found in Table 12

In the literature, there are two authors who published such a Knuthian analysis of Quicksort. [Knu98, Section 5.2.2] and [Sed75, Chapter 3] both study the same `MIX` implementation of classic Quicksort with explicit stack management, tail recursion elimination and Insertionsort for small subfiles (see Section 3.4.3). The overall runtime can be expressed in a handful of combinatorial quantities, namely

A number of partitioning steps,
B number of swaps,
C number of comparisons,
D number of insertions (during insertion sort),
E number of key movements (during insertion sort),
S number of stack pushes .

The total `MIX` runtime is then

$$24A + 11B + 4C + 3D + 8E + 9S + 7n \ .$$

Inserting the expected values for $A, \dots, S$ and setting the Insertionsort threshold to $M = 1$, we get the `MIX` runtime of a program similar to Algorithm 1. Its average runtime is

$$11.\overline{6}(n+1)\mathcal{H}_n + 5.5\overline{7}n - 17.2\overline{5} \ .$$

[9] In fact, Hennequin gives a much more general result for Quicksort with median of k pivot selection. Thereby, he could show that pivot sampling reduces variance.

3.4 Variants of Quicksort

The abstract description of Quicksort leaves room for many possible variants. The following sections discuss the most prominent ones among those. Improvements of different categories can sometimes be combined successfully.

It is remarkable that most variants discussed below were already suggested in HOARE's original publication of Quicksort [Hoa62] in the early 1960s. However, it took the algorithms community several decades to precisely quantify their impact. And as new variants keep appearing from time to time, their study is an ongoing endeavor.

3.4.1 Choice of the Pivot

The abstract description of Quicksort from Section 3.1 leaves it open, how to choose the pivot — or the $s-1$ pivots in the generalized version. Classic Quicksort chooses one fixed position from the array as pivot element — namely the last element for Algorithm 1. By applying some more elaborate scheme, we can influence the distribution of the *rank* of the pivot elements, which in turn affects performance. This section gives a mainly chronological review of suggested pivot selection methods and their impact on classic Quicksort.

In his seminal article [Hoa62] on Quicksort, HOARE analyzes classic Quicksort and observes that balanced recursion trees contribute the least cost. A perfectly balanced tree results if the pivot is the *median* of the current list. Therefore, HOARE suggests to choose as pivot "the median of a small random sample of the items in the segment", which gives a maximum likelihood estimate of the median of the whole list. However, HOARE attests: "It is very difficult to estimate the savings which would be achieved by this, and it is possible that the extra complication of the program would not be justified."

The idea is put into practice by SINGLETON in [Sin69]. He proposes a Quicksort variant that uses the median of three elements, namely the first, the last and the middle element of the list. SINGLETON reports speedups in runtime of about 25 %.

The first analytical approach to investigate the impact of this pivot sampling strategy is undertaken by VAN EMDEN in [van70]. He assumes a Quicksort variant that selects as pivot the median of a sample of odd size $k = 2t + 1$, for some constant $t \in \mathbb{N}$. Using a purely information theoretic argument, he derives that the expected number of comparisons used by Quicksort with median of k to sort a random permutation of $[n]$ is

$$\alpha_k n \ln n + o(n \log n) \qquad \text{with} \quad \alpha_k = 1 \Big/ \sum_{i=1}^{k+1} \frac{(-1)^{i+1}}{i} \,. \tag{3.2}$$

VAN EMDEN also notes that $\lim_{k\to\infty} \alpha_k = 1/\ln 2$, which proves that Quicksort with median of k is asymptotically optimal w. r. t. comparisons if k is allowed to grow.

The situation that k depends on n is analyzed in [FM70]. Therein FRAZER and MCKELLAR propose Samplesort, which initially selects a sample of size $k = k(n)$, sorts this sample — potentially recursively by Samplesort — and then select its median as pivot. What makes Samplesort different from Quicksort with median of k is that the sorted halves below and above the median are "passed down" to the recursive calls. Then, the recursive calls pick the median of their half, and so on. That way, the sample size is halved in each step and recursive calls need not sort their sample. When the sample is consumed, Samplesort continues as classic Quicksort. Considering the recursion tree, Samplesort ensures that the first $\lfloor \log_2 k(n) \rfloor$ levels are complete.[10]

Sedgewick's PhD thesis [Sed75] is considered a milestone in the study of Quicksort. In particular, it resolved many open problems regarding the analysis of pivot selection schemes. In [Sed75, Chapter 7], a detailed analysis of Samplesort is given. For instance, the optimal sample size is asymptotically $k(n) \approx \frac{n}{\log_2 n}$ and for that sample size, Samplesort is asymptotically optimal w. r. t. the number of comparisons.

Further, Sedgewick derives the expected number of comparisons *and* swaps for Quicksort with median of constant k using generating functions and techniques quite similar to Section 4.2.2. He finds[11]

$$\text{comparisons} \qquad \frac{1}{\mathcal{H}_{2t+2} - \mathcal{H}_{t+1}} n \ln n + \mathcal{O}(n), \tag{3.3}$$

$$\text{swaps} \qquad \frac{4 + 11\frac{t+1}{4t+6}}{\mathcal{H}_{2t+2} - \mathcal{H}_{t+1}} n \ln n + \mathcal{O}(n)\,. \tag{3.4}$$

Moreover, Sedgewick provides an exact Knuthian analysis of a low level implementation of Quicksort with median of three [Sed75, Chapter 8]. These important results are also contained in [Sed77b].

In [Gre83], Greene introduces *diminished search trees*, which form the search tree analog of Quicksort with median of k for constant k. This allows to study Quicksort with median of k using the correspondence described in Section 3.5.1.

In his thesis [Hen91], Hennequin analyzes a more general Quicksort variant, which combines pivot sampling and multi-pivot partitioning. Hennequin's work relies on heavy use of generating functions and is discussed in much more detail in Section 3.5. The most notable contribution to the study of pivot sampling is the computation of higher order moments, which is also contained in [Hen89]. The exact expressions are quite lengthy and the interested reader is referred to [Hen89, Section 4.3]. I only note that all standard deviations are in $\mathcal{O}(n)$ as for classic Quicksort. Hence, the centralization argument from Section 3.3.3 also applies for Quicksort with median of k.

[10] There is some inconsistency in the use of "Samplesort" in the literature. Originally, Samplesort as introduced in [FM70] explicitly constructs a binary search tree. Translated to Quicksort, this means that it once fixes a sample of size $k(n)$ and passes the sorted halves of the sample down. That's how it is described in [FM70], [Tan93] and [Sed75]. In later usages, Samplesort seems to imply the use of $s - 1 > 1$ pivots, which are selected from a sample of size $k(n)$, e. g. in [SW04], [BLM$^+$91], [LOS09]. The main difference between these algorithms is how the sample size changes in recursive calls. Classic Samplesort halves the sample size, whereas the newer interpretations use a sample of size $k(n)$ at every step. In this thesis, I use Samplesort to refer to the original algorithm.

[11] Both Eqs. (3.2) and (3.3) describe the same quantity. Equating the leading terms gives the noteworthy equivalence $\mathcal{H}_{2k} - \mathcal{H}_k = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \cdots + \frac{1}{2k-1} - \frac{1}{2k}$.

A different flavor of pivot selection is proposed by van Emden in [van70]: *bounding interval*, a. k. a. *adaptive partitioning*[12]. Here, the idea is to start partitioning without having chosen a fixed pivot. Instead, we only maintain an interval of values, from which the pivot is finally chosen. Partitioning proceeds as usual as long as it is clear to which partition an element belongs. If an element x falls inside the current pivot interval, the interval is shrunk, such that x falls on one side. From the point of view of the analysis, adaptive methods are problematic, as they do not preserve randomness for sublists. Van Emden numerically determines the leading term of the number of comparisons of his algorithm to be $1.6447n \log_2 n$, which sounds promising. However, Sedgewick argues in [Sed75, Chapter 6] that adaptive methods increase the costs of inner loops, which overcompensates the savings in comparisons for overall runtime.

In [BM93], Bentley and McIlroy try to design an optimal Quicksort based sorting method for use in programming libraries. Apart from interesting suggestions to deal with equal elements — which will be covered in Section 3.4.2 — the authors also propose a revised pivot sampling scheme. The key idea is to use a cheap surrogate for the real median of a 'large' sample, the *pseudomedian of nine*. This statistic takes nine elements from the list and divides them in three groups of three. Then, we choose the median of the three medians from the three groups as pivot. The expected number of comparisons Quicksort needs when equipped with this pivot selection strategy is computed in [Dur03] to be

$$\tfrac{12600}{8027} n \ln n + \mathcal{O}(n) \quad \approx \quad 1.5697 n \ln n + \mathcal{O}(n)\ .$$

Samplesort started off with the idea of using the median of a sample whose size k depends on n. However, it only uses this size in the very first partitioning step. It seems natural to ask how $k = k(n)$ should be chosen, if we randomly choose a sample of size $k(n)$ in each partitioning step, where n then is the size of the sublist. This question is addressed in [MT95] and [MR01]. Roughly speaking, $k(n) = \Theta(\sqrt{n})$ provides the best overall performance, even if swaps are taken into account [MR01]. Martínez and Roura also give the optimal constant in front of the square root.

For the most part of this thesis, I study basic algorithms which choose their pivots from fixed positions out of the array. However in Chapter 9, I will consider sampling the pivots from a constant size sample.

3.4.2 Implementation of the Partitioning Step

The abstract description of Quicksort from Section 3.1 only defines how the array should look like after it has been partitioned around the pivot. It does not give us clues how to arrive there. In his PhD thesis [Sed75], Sedgewick studies and compares several in-place partitioning methods. I chose the most effective of those for my classic Quicksort given in Algorithm 1. The underlying partitioning scheme is Hoare's crossing pointers technique, whose invariant is depicted in Figure 2 on page 26. The partitioning method of Algorithm 2 is given in the classic textbook [CLRS09] and uses a different scheme.

[12] Van Emden calls his variant bounding interval method, Sedgewick refers to it as adaptive partitioning.

Algorithm 2. Quicksort variant from [CLRS09, Chapter 7]. It uses a particularly simple partitioning scheme, which is *not* based on HOARE's crossing pointers technique.

```
QuicksortCLRS($A$, $left$, $right$)
if $left < right$
    $p := A[right]$; $i := left - 1$
    for $j := left, \dots, right - 1$
        if $A[j] \leqslant p$
            $i := i + 1$
            Swap $A[i]$ and $A[j]$
        end if
    end for
    $i := i + 1$
    Swap $A[i]$ and $A[right]$
    QuicksortCLRS($A$, $left$ , $i - 1$)
    QuicksortCLRS($A$, $i + 1$, $right$)
end if
```

It is appealingly short and probably easier to implement than Algorithm 1. However, it uses *three times* as many swaps as Algorithm 1, which probably renders it uncompetitive.

Apart from different partitioning schemes, there are also little tricks to make the innermost loops even more efficient. One such idea is the use of *sentinels*, first proposed by SINGLETON in [Sin69]. A pointer scanning the current sublist must be stopped before it leaves its range. Cleverly placed sentinel elements at the ends of the sorting range will guard the pointers from running out of range without an explicitly check of the bounds. We only have to make sure that the key comparisons done anyway fail when the range is left.

Algorithm 1 incorporates this idea. The only additional work is to put an element '$-\infty$' in $A[0]$ before sorting. It suffices if $A[0] \leqslant A[i]$ for all i, strict inequality is not needed. Then, pointer j will stop at $A[0]$ for good, as for any pivot p, $A[0] \leqslant p$. Similarly, i will stop at the pivot p, which is chosen to be the last element of the list. This constitutes the base case of an inductive correctness proof. To complete it, note that the sentinel property is preserved for recursive calls: We either have $left = 1$, in which case $A[0]$ will be the sentinel again, or $left = p + 1$, one cell right of the final position of the pivot. Then, the pivot of the current phase will serve as sentinel of the upcoming partitioning step. This works, as all elements right of p are $\geqslant p$ by the end of the partitioning step.

Further low level tricks are investigated by SEDGEWICK in [Sed75] and [Sed78], which might be worth considering for practical implementations.

Pitfall: Equal Elements Even though partitioning is a rather elementary problem, some care has to be taken to avoid delicate pitfalls. A typical problem results from the presence of *equal keys*. I will assume distinct elements for the rest of this thesis, but this section puts special emphasis on their existence because some well-known partitioning methods perform poorly in this situation.

Recall the simple partitioning scheme from [CLRS09] shown in Algorithm 2. This variant degrades to *quadratic average complexity* on lists where keys are taken from a set of constant cardinality, i. e. where many duplicate keys exist. For demonstration, consider a list where *all* elements are equal. On this list, the comparison in line 4 always yields true, so i runs all the way up to *right* and we get the worst case for recursive calls: One is empty, the other contains $A[1..n-1]$.

In light of this, it is impressive that Algorithm 1 remains linearithmic for this same list. This is easily seen by inspecting the code, but it might be more amusing to tell this in form of an anecdote:

A tale of "Premature Optimization"

At first sight, there seems to be an immediate improvement for Algorithm 1 if many duplicate keys are to be expected, especially if all elements in the array are equal: The inner while loops in lines 5 and 6 will always terminate immediately and the swap in line 7 is executed for every (nested) pair of indices. In fact, for a single partitioning step, this is the worst case with respect to the number of executed swaps … can't we do better than that?

When I first made the above observation, I proudly tried the following: Instead of checking for strict inequality in lines 5 and 6, we check for $\leqslant p$ and $\geqslant p$, respectively. The algorithm remains correct and indeed on a list with all elements equal only the unconditional swap in line 9 is done. Hooray!

Yet, the joy was not to last: The clever "improvement" causes the first while loop to scan over the entire array. This results in a final value of $i = \mathit{right}$ and we always get the worst possible division: One of the subfiles is empty, whereas the other contains all but one element. Consequently, we get overall quadratic runtime. The original method in Algorithm 1 does indeed perform many swaps, none of which would be necessary in this case. However, the pointers i and j always meet in the middle, so we always get the optimal division. In fact, the number of comparisons performed is the best case for Algorithm 1. This effect has been known for quite some time, e. g. it is reported in [Sin69].

So, can we really do better than Algorithm 1 on lists with many duplicate keys? Well, yes, by using a different partitioning scheme, e. g. the one from [BM93] specifically designed to handle these inputs … but by tiny clever tweaks of Algorithm 1? I teeth-gnashingly admit: No.

Algorithm 3. Quicksort with simple three-way partitioning from [SW11, page 299]. Note the resemblance to Algorithm 8; in fact YAROSLAVSKIY's algorithm can be seen as improved version of this algorithm's partitioning scheme.

```
ThreeWayQuicksort(A, left, right)
if left < right
    p := A[left]
    ℓ := left; k := left + 1; g := right
    while k ⩽ g
        if A[i] < p
            Swap A[ℓ] and A[i]
            ℓ := ℓ + 1; k := k + 1
        else if A[i] > p
            Swap A[k] and A[g]
            g := g − 1
        else i := i + 1 end if
    end while
    ThreeWayQuicksort(A, left , ℓ − 1)
    ThreeWayQuicksort(A, g + 1, right)
end if
```

Three-Way Partitioning

Algorithm 1 remains linearithmic on average for lists with duplicate keys, so our classic puts up a good fight. Still the many unnecessary swaps seem suboptimal and in lists with many elements equal to the pivot, we might want to directly exclude all of them from recursive calls. Algorithm 1 only excludes one copy of the pivot. As the case of equal elements in the list appears frequently in practice it might pay to include special handling of this case.

This special handling consists in doing a *three-way partitioning* — separating elements that are strictly less than the pivot from those strictly larger *and* from all elements equal to the pivot. A surprisingly simple method to do so is shown in Algorithm 3, originally taken from [SW11]. Therein it is also shown that Algorithm 3 uses a linear number of comparisons if elements are taken from a constant size set. A disadvantage of Algorithm 3 is that for distinct elements, it uses much more swaps than Algorithm 1 — in fact roughly *six times* as many swaps — and some more comparisons, namely 50 % more than Algorithm 1 for a random permutation. So, the question arises whether we can find an algorithm that smoothly adapts to both input types.

In fact, such a Quicksort variant exists: Algorithm 4. This slightly more involved three-way partitioning scheme was proposed in [BM93]. The main idea of Algorithm 4 is

Algorithm 4. Quicksort with Bentley and McIlroy's three-way partitioning method proposed in [BM93].

```
QuicksortBentleyMcIlroy($A$, $left$, $right$)
if $left < right$
    $p := A[left]$
    $i := left$; $j := right + 1$
    $\ell := left$; $g := right + 1$
    while true
        while $A[i] < p$
            $i := i + 1$
            if $i == right$ then break inner loop end if
        end while
        while $A[j] > p$
            $j := j - 1$
            if $j == left$ then break inner loop end if
        end while
        if $i \geqslant j$ then break outer loop end if
        Swap $A[i]$ and $A[j]$
        if $A[i] == p$
            $\ell := \ell + 1$
            Swap $A[j]$ and $A[\ell]$
        end if
        if $A[j] == p$
            $g := g - 1$
            Swap $A[j]$ and $A[g]$
        end if
    end while
    Swap $A[left]$ and $A[j]$
    $i := i + 1$; $j := j - 1$
    for $k := left + 1, \dots, \ell$
        Swap $A[k]$ and $A[j]$ end for
        $j := j - 1$
    end for
    for $k := right, \dots, g$
        Swap $A[k]$ and $A[i]$ end for
        $i := i + 1$
    end for
    QuicksortBentleyMcIlroy($A$, $left$ , $j$ )
    QuicksortBentleyMcIlroy($A$, $i$ , $right$)
end if
```

to move elements equal to the pivot to the left respectively right end of the array during partitioning. At the end of the partitioning step, they are swapped into the middle and skipped for recursive calls. By that, Algorithm 4 does not incur additional swaps and only a few more comparisons for lists with distinct elements than Algorithm 1. For lists with many duplicates however, it pays to exclude all elements equal to the pivot in a single partitioning step.

3.4.3 Treatment of Short Sublists

We convinced ourselves in Section 3.3 that Quicksort is efficient for medium and large list sizes. For very small sizes $n \leqslant M$, say $M = 15$, however, Quicksort is comparatively slow. At first sight one might doubt why this is relevant — if people really need to sort short lists, they had better use a specialized sorting method. The problem is that Quicksort's recursive nature leads to many *recursive* calls on such small lists, even if the initial list to be sorted is huge! Therefore, the savings achievable by switching to a cheaper sorting method for small sublists should not to be underestimated.

This has already been noticed by Hoare in [Hoa62], but without an explicit recommendation what to do with lists of size $\leqslant M$. Singleton then uses straight Insertionsort for small sublists in [Sin69], which he determined empirically to be a good choice. This choice was later refined by Sedgewick in [Sed75] as follows. As Insertionsort is very efficient on almost sorted lists, Sedgewick proposes to first *ignore* lists of size $\leqslant M$ and later do a *single* run of Insertionsort over the whole list. This significantly reduces the overhead of the method.

The variant of special treatment of short sublists can be combined with many other optimizations without interference as most variants aim to improve behavior on large lists. It is also easily incorporated in recursive analyses of Quicksort's costs. Nevertheless, I do not include this optimization in my Quicksort variants as the additional parameter M clutters the results and computations. I merely note that the adaption of the analyses is straight-forward.

3.4.4 Multi-Pivot Quicksort

In the abstract description of Quicksort, I already included the option to generalize Quicksort to using $s - 1$ instead of just one pivot element. Whereas the abstract idea to do so seems natural, multi-pivot Quicksort has been studied by far not as thoroughly as classic Quicksort in the literature. Especially when it comes to concrete in-place implementations of multi-pivot partitioning methods, only a few sources are available.

To the author's best knowledge, [Sed75] contains the first published implementation of a dual pivot Quicksort. It is given as Algorithm 7 on page 52. Sedgewick analyzes in detail the number swaps[13] Algorithm 7 needs to sort a random permutation of the set $[n]$,

[13] Sedgewick also gives a term for the expected number of comparisons used by Algorithm 7. However, the leading term is below the information theoretic lower bound of Section 2.3, so most probably a typing error has happened there.

Algorithm 5. The general reference Quicksort considered in [Hen91].
This implementation is based on linked lists. We denote by $\otimes$ the concatenation of lists and identify elements with the list containing only this element. Remove(L, m) removes the first m elements from list L and returns them as list, i. e. we split L after the mth entry. The BinarySearch procedure uses an implicit perfect binary search tree (called *decision tree* in [Hen91]). Note that the for any fixed s, we can statically unfold BinarySearch (cf. preprocessor macros in the C programming language).

```
HennequinsReferenceQuicksort($L$)
    // Sort linked list $L$ using $s-1$ equidistant pivots from a sample of $k := s \cdot t + (s-1)$ elements
    // and using a special purpose SmallListSort for lists of lengths $\leqslant M$ (with $M \geqslant k-1$).
    if length($L$) $\leqslant M$ then return SmallListSort($L$)
    else
        sample := Sort(Remove($L$, $k$))
        for $i := 1, \ldots, s-1$
            $L_i$ := Remove(sample, $t$)
            $p_i$ := Remove(sample, $1$)
        end for
        $L_s$ := Remove(sample, $t$) // sample is empty now
        while $L$ is not empty
            $x$ := Remove($L$, $1$)
            $i := \text{BinarySearch}_{[p_1 \cdots p_{s-1}]}(x)$
            $L_i := L_i \otimes x$
        end while
        for $i := 1, \ldots, s$
            $L_i$ := HennequinsReferenceQuicksort($L_i$)
        end for
        return $L_1 \otimes p_1 \otimes \cdots \otimes L_{s-1} \otimes p_{s-1} \otimes L_s$
    end if

    $\text{BinarySearch}_{[p_i]}(x)$
        return if $x < p_i$ then $i$ else $i+1$
    $\text{BinarySearch}_{[p_i p_{i+1}]}(x)$
        return if $x < p_i$ then $i$ else $\text{BinarySearch}_{[p_{i+1}]}(x)$
    $\text{BinarySearch}_{[p_i \cdots p_{i+\ell-1}]}(x)$        // $(l \geqslant 3)$
        $m := \lfloor \frac{\ell+1}{2} \rfloor$
        if $x < p_m$ then return $\text{BinarySearch}_{[p_i \cdots p_{m-1}]}(x)$
        else              return $\text{BinarySearch}_{[p_{m+1} \cdots p_{i+\ell-1}]}(x)$
```

Algorithm 6. Triple-pivot Quicksort by Tan from [Tan93, Figure 3.2]. It uses a procedure Partition which partitions the array using the first element as pivot and returns its final index. This procedure can be implemented similar to Algorithm 1. It is convenient to let the algorithm skip sublists of size $\leqslant 3$. A final run of Insertionsort can then be used, as described in Section 3.4.3. The idea of this algorithm easily extends to $s = 2^r$ for $r \in \mathbb{N}$.

```
MultiPivotQuicksortTan($A$, $left$, $right$)
if $right - left \geqslant 2$
    $p_1 := A[left]$; $p_2 := A[left + 1]$; $p_3 := A[left + 2]$
    Sort pivots s. t. $p_1 \leqslant p_2 \leqslant p_3$
    Swap $p_3$ and $A[right]$    // Move $p_3$ out of the way
    $i_2 :=$ Partition($A$, $left + 1$, $right - 1$)    // Partition whole list around $p_2$
    Swap $A[right]$ and $A[i_1 + 1]$    // Bring $p_3$ behind $p_2$
    $i_1 :=$ Partition($A$, $left$, $i_2 - 1$)    // Partition left part around $p_1$
    $i_3 :=$ Partition($A$, $i_2 + 1$, $right$)    // Partition right part around $p_3$
    MultiPivotQuicksortTan($A$, $left$, $i_1 - 1$)
    MultiPivotQuicksortTan($A$, $i_1 + 1$, $i_2 - 1$)
    MultiPivotQuicksortTan($A$, $i_2 + 1$, $i_3 - 1$)
    MultiPivotQuicksortTan($A$, $i_3 + 1$, $right$)
end if
```

namely $0.8n \ln n + \mathcal{O}(n)$. This lead the author to the conclusion that this algorithm is not competitive with classic Quicksort, which gets along with only $\frac{1}{3} n \ln n + \mathcal{O}(n)$ swaps (see Section 3.3.2).

In [Hen91], Hennequin studies the generic list based Quicksort given in Algorithm 5. Even though this algorithm is not in-place, many properties can be transferred to in-place implementations of multi-pivot Quicksort and much of the methodology of analysis carries over. Hennequin determines the expected number of comparisons as function in s. Considering this quantity, his conclusion is that multi-pivot Quicksort in general, and dual pivot Quicksort in particular, are not promising variations of Quicksort.

However — as we will see throughout Chapter 4 and discuss in Section 4.4.1 — Hennequin makes some simplifying assumptions by considering Algorithm 5. These assumptions are not fulfilled by the dual pivot Quicksort variants Algorithms 7 and 8.

Tan considers multi-pivot Quicksort in his PhD thesis [Tan93] for $s = 2^r$ with $r \in \mathbb{N}$. Tan's algorithm is shown for $s = 4$ in Algorithm 6 and works like this: First, it selects $s - 1$ pivot elements and sorts them by a primitive sorting method. Then, the current sublist is partitioned around the median of $s - 1$ pivots using an ordinary binary partitioning method, e. g. as in Algorithm 1. Afterwards the quartiles of the pivots are used to partition the two parts of the list into four parts. This process is continued until all pivots have been used and we end up with s partitions. On these, we recursively invoke the algorithm. That way, one can reuse the efficient partitioning strategies for classic Quicksort for multi-pivot Quicksort.

TAN's algorithm is similar to Samplesort of [FM70] described in Section 3.4.1. However, Samplesort *once* chooses a sample of size $k(n)$ depending on n and once this sample is exhausted, Samplesort behaves like ordinary Quicksort. In contrast, TAN's algorithm chooses a 'sample' of fixed size $s - 1 = 2^r - 1$ in *every* partitioning step. Viewed from this perspective, TAN's multi-pivot Quicksort resembles classic Quicksort with median of $k = s - 1$ pivot selection. However, the next few partitioning steps are predetermined and use the corresponding order statistics of the sample as pivot.

TAN computes the expected number of comparisons required by Algorithm 6 to sort a random permutation of $[n]$. He arrives at the recurrence

$$C_n = 12 \sum_{j=1}^{n-2} \frac{(n-j-1)(n-j-2)}{n(n-1)(n-2)} C_j + 2n - \tfrac{10}{3} .$$

Since TAN is mainly interested in limiting distributions of costs, he confines himself to empirically determine C_n from computed points via fitting. That way, he found $C_n \approx 1.8449 n \ln n + \mathcal{O}(n)$. Note that this is very close to the expected number of comparisons HENNEQUIN computes for Algorithm 5 with $s = 4$ and $t = 0$, namely $1.846 n \ln n + \mathcal{O}(n)$ (see Section 3.5.4). Therefore, we might take TAN's algorithm as an in-place implementation of Algorithm 5 for $s = 2^r$.

Note further that TAN's algorithm with $s = 4$ is closely related to classic Quicksort with median of three pivot sampling. Both methods start by partitioning around the median of three elements, but whereas TAN then also uses the remaining two elements as pivots for two more partitioning calls, classic Quicksort immediately picks three new elements. Looking at the expected number of comparisons, the latter seems to be the better choice: Classic Quicksort with median of three needs $1.\overline{714285} n \ln n + \mathcal{O}(n)$ comparisons. So it saves every 14th comparison TAN's algorithm does — *and* it is simpler.

It is noteworthy that the analyses done for the above algorithms show only minor if any savings due to multi-pivot approaches. They might not justify the additional complexity in practice. Even more so, as we have seen promising variations of Quicksort in the preceding sections, which are easier to handle.

This might explain the relative low interest in multi-pivot Quicksort. As we will see in the remaining chapters of this thesis, this rejection might have been more due to an unlucky selection of implementations than that it is due to inherent inferiority of multi-pivot partitioning.

3.5 Hennequin's Analysis

In his PhD thesis *"Analyse en moyenne d'algorithmes : tri rapide et arbres de recherche"* [Hen91], ("Average case analysis of algorithms: Quicksort and search trees"), Hennequin considers Quicksort with several pivots. He analyzes the linked-list based "reference Quicksort" shown in Algorithm 5 on page 40 which incorporates several of the variants mentioned in Section 3.4 in a parametric fashion: The current list is divided into s partitions around $s-1$ pivots for an arbitrary, but constant $s \geqslant 2$. The pivot elements are chosen equidistantly from a sample of $k = s \cdot t + (s-1)$ elements for fixed constant $t \geqslant 0$, i. e. the ith largest pivot p_i is chosen to be the $i(t+1)$-st largest element of the sample ($1 \leqslant i < s$). Moreover, small (sub) lists of size $\leqslant M$ are treated specially by a different sorting method SmallListSort.

I consider Hennequin's thesis a major contribution to our understanding of Quicksort. Yet it is only available in French and to the author's knowledge, there is no other publication that describes his analysis of general Quicksort. Note in particular, that [Hen89] only covers the results for the case $s = 2$. Therefore, the rest of this section is devoted to recapitulate the parts of [Hen91] relevant to the this thesis in some more detail. That other seminal contributions have been discussed rather briefly above should not be considered a depreciation of that work. I merely think, the reader is more willing to read those him- or herself unless his or her French is better than mine.

To study Algorithm 5, Hennequin uses the the theory of combinatorial structures introduced in [FS09] as *symbolic method*. To apply this, we need to translate the execution of Quicksort to a recursive decomposition of an equivalent class of combinatorial structures.

3.5.1 Equivalence of Quicksort and Search Trees

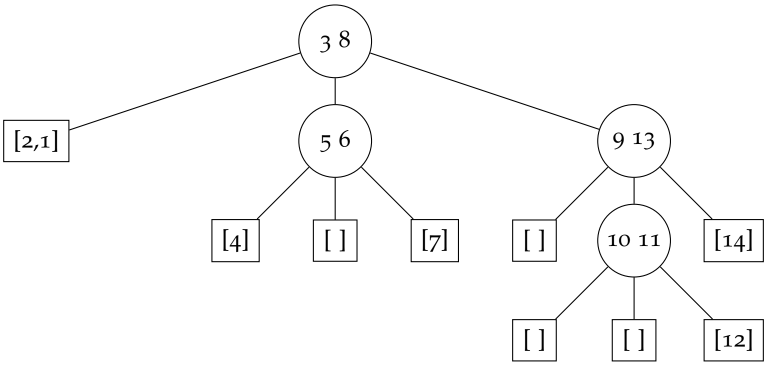
The recursion tree of a recursive procedure is defined inductively: For a call with 'base case' arguments, i. e. a call that does not recurse at all, the recursion tree is a single leaf. Otherwise, there are $c > 0$ recursive calls. Let $T_1, \ldots, T_c$ be the corresponding recursion trees. Then, the recursion tree for the whole call is a new root with children $T_1, \ldots, T_c$, in the order in which the calls appear in the procedure. By this, we assign to each terminating procedure call a rooted ordered tree of finite depth and degree.

For Quicksort, each call causes either 0 or s recursive calls. Hence, its recursion trees are s-ary trees, i. e. all inner nodes have exactly s children. Each inner node corresponds to a partitioning step with $s-1$ pivots and each leaf to a small list of $\leqslant M$ elements. Thus, we can label internal nodes with the list of pivot elements and leaf nodes with their $\leqslant M$-element-list. By definition of the partitioning process, we have the following fact.

Fact 3.1: A recursion tree of Quicksort fulfills the *search tree property*, i. e. for all inner nodes x holds: When T_l and T_r are two subtrees of x where T_l is left of T_r, then all labels in T_l are smaller than all labels in T_r. Moreover, there is an element p in the label of x such that p lies between all labels of T_l and the ones of T_r. □

We obtain the same tree if we successively insert elements in the order in which they are chosen as pivots into an initially empty search tree. In general, the order in which elements are chosen as pivots can be rather complicated. As Algorithm 5 always chooses the first elements of a list as pivots, its recursion tree coincides with the search obtained by successively inserting the list elements from left to right.

Example 3.2: The ternary search tree resulting from Algorithm 5 with $s = 3$, $t = 0$ and $M = 2$ on the list $[3, 8, 2, 1, 5, 6, 7, 4, 9, 13, 14, 10, 12, 11]$



3.5.2 Combinatorial Analysis of Quicksort

The equivalence established in the Section 3.5.1 allows to reason about the number of comparisons needed by Quicksort by considering its recursion trees. Let us first consider the classic Quicksort ($s = 2$).

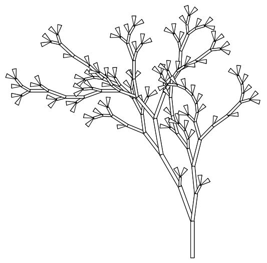


Figure 3: Typical random binary search tree with 100 nodes drawn according to the binary search tree model. This tree is taken from [Knu11, ex. 7.2.1.6-124].

Then, the labelled recursion trees are plain binary search trees. Each element of a current sublist experiences one comparison, namely with the pivot. Summing all comparisons involving the single element x across all recursive calls is then equivalent to asking how many partitioning steps there are, where x participates. All elements are either passed down in the left or right subfile — except for the pivot. Hence, the number of partitioning steps for x equals the *depth* of x in the associated recursion tree, where the depth of x is the number of edges on the path from the root to x.

Summing over all elements x yields the overall number of comparisons for sorting the list with classic Quicksort, which is exactly the *internal path length* (for binary trees). This correspondence is mentioned by Knuth [Knu98, Section 6.2.2], even though it is not exploited to analyze Quicksort, there.

The main simplification in analyzing search trees instead of Quicksort itself lies in the *static* nature of the trees — we do not have to reason about the possibly intricate dynamics of a partitioning step. Instead, we analyze the static properties of static structures. A probability distribution of inputs for the algorithm translates to a distribution over structures. For the expected number of comparisons for classic Quicksort, we compute the expected path length of random binary search trees with n nodes. The random permutation model then translates to the following distribution over trees: The probability $\Pr[T]$ of a search tree T is $\Pr[T] := \frac{N(T)}{n!}$, where $N(T)$ is the number of permutations such that successive insertions into an initially empty binary search tree yield T.

Martínez [Mar92, page 35] shows that these probabilities can be defined recursively over the tree structure:

$$\Pr[T] = \begin{cases} 1 & \text{if } |T| = 0 \\ \frac{1}{|T|} \cdot \Pr[T_l] \cdot \Pr[T_r] & \text{otherwise} \end{cases},$$

where $|T|$ denotes the number of nodes in T and T_l and T_r are the left and right subtrees of T, respectively. This decomposition allows to set up a differential equation for the generating function of the expected path lengths. The expected path length of a random binary tree under the binary search tree model is

$$2n\mathcal{H}_n - 4n + 2\mathcal{H}_n \,. \qquad \begin{array}{rl} & \text{[Mar92, page 36]} \\ \text{or} & \text{[Knu98, page 431]} \end{array}$$

The expected number of comparisons for Algorithm 1 is $2n\mathcal{H}_n - \frac{8}{3}n + 2\mathcal{H}_n + \mathcal{O}(1)$, see Table 1. The reason for the deviation is that Algorithm 1 does two extra comparisons per partitioning step because of the sentinel-trick (see Section 3.4.2).

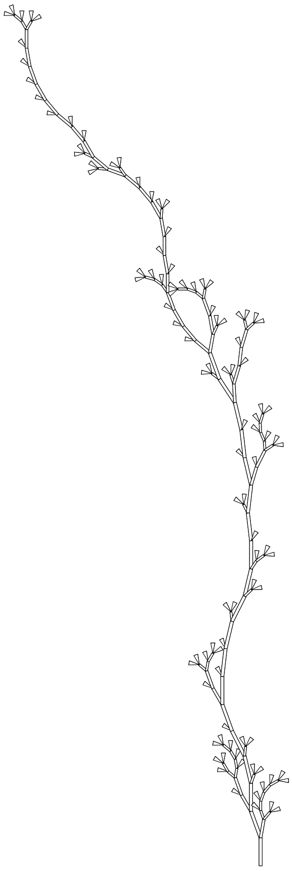


Figure 4: Typical random binary tree with 100 nodes drawn according to uniform distribution.
This tree is taken from [Knu11, ex. 7.2.1.6-124].

For multi-pivot Quicksort, we may need more than one comparisons per element and partitioning step. In fact, for each element x in a partitioning step, Algorithm 5 conducts an unsuccessful search for x in a perfect binary search tree of the pivots. The number of comparisons needed for this search is not constant — it depends on the binary search tree and x. Yet, it is known that perfect binary search trees have all their leaves on two consecutive levels (e. g. [Knu98, exercise 5.3.1-20]), so the number will not differ by more than one.

As long as we only consider the *expected* number of comparisons needed by Quicksort, we can replace the actual costs for the unsuccessful searches by their expectation — because of the linearity of the expectation and since the total costs depend linearly on the costs per search. The expected number of comparisons needed for an unsuccessful search in a perfect binary search tree with $N = s - 1$ nodes is given in [Knu98, eq. 6.2.1–(4)]:

$$k + 2 - 2^{k+1}/(N+1) \qquad \text{for } k = \lfloor \log_2 N \rfloor \,. \tag{3.5}$$

When we incorporate the selection of pivots equidistantly from a sample of $k = st + (s-1)$ elements, we get a different distribution on the recursion trees. This distribution assigns more weight to more balanced trees. For example, for $t > 0$, degenerate search trees, e. g. linear lists, can no longer occur as recursion trees.

Greene introduces and analyzes a generalization of binary search trees in [Gre83, Section 3.2] called *diminished trees*, which turn out to be equivalent to recursion trees of classic Quicksort ($s = 2$) with median of k. A diminished tree is essentially a binary search tree, which is somewhat reluctant in creating new inner nodes. The inner nodes are ordinary nodes, but its leaves can store up to $k-1$ keys for an uneven constant $k \geqslant 1$.

To insert a new key c, we traverse the search tree until we reach a leaf. If this leaf has a free cell, it simply stores c and we are done. Otherwise, the leaf already contains $k-1$ keys. Then, we create a new inner node for the *median* of the elements (including c) and two leaves holding the $\frac{k-1}{2}$ elements that are smaller respectively larger than the median. These leaves become the children of the new inner node. By this delayed creation of inner nodes, diminished trees always take a locally balanced shape.

Hennequin combines all these generalizations in his analysis. However — in order to avoid the non-uniform probability distributions over recursion trees — Hennequin translates the operations back to permutations. In the following, I assume basic familiarity with the symbolic method described in [FS09]. I also adapted Hennequin's notation to match that of [FS09].

He gives the following recursive construction for the combinatorial class of permutations induced by Algorithm 5 [Hen91, Théorème III.1]:

$$\mathcal{S} = \mathrm{T}_M \Big[\circ_k \big(\underbrace{\Delta_t(\mathcal{S}) \star \cdots \star \Delta_t(\mathcal{S})}_{s \text{ times}} \big) \Big] + \mathrm{R}_M(\mathcal{S}) \,, \tag{3.6}$$

where $\mathcal{S}$ denotes the labelled combinatorial class of permutations [FS09, Example II.2], $+$ is the disjoint union of classes and $\star$ denotes the *partitional* or *labelled product* [FS09, Definition II.3]. T_M, R_M, $\circ_k$ and Δ_t are special unary operators on combinatorial classes:

$$\begin{array}{ll} \text{Truncation at } M & \mathrm{T}_M(\mathcal{C}) := \big\{ c \in \mathcal{C} : |c| > M \big\} \\ \text{Remainder up to } M & \mathrm{R}_M(\mathcal{C}) := \big\{ c \in \mathcal{C} : |c| \leqslant M \big\} \\ \text{Rooting of order } k & \circ_k(\mathcal{C}) := \mathcal{S}_k \times \mathcal{C} \\ \text{Deletion of order } t & \Delta_t(\mathcal{S}_t \times \mathcal{C}) := \mathcal{C} \end{array}$$

The deletion operator of order t removes the first t components of a structure. It is not well-defined for arbitrary structures since the notion of "the first t components" might not make sense for some structures. We only apply it to $\mathcal{S}$ which can always be suitably decomposed, though, so there is no reason to panic, here.

The construction in eq. (3.6) can be motivated by reversing the partitioning step: Each permutation is either of size $> M$ or $\leqslant M$; the latter case being the base case (Algorithm 5 terminates). A large permutation σ of length $n > M$ is now formed as follows: The rooting operator says that σ decomposes into $\sigma = \tau\,\sigma'$ such that τ has length k. τ corresponds to the sample of k elements we remove in line 3 of Algorithm 5. σ' is the part of the permutation constructed from smaller permutations.

From the sample, we pass down t elements into each partition (line 5). But we already have those elements in τ. So, when constructing σ from smaller permutations, we remove these first t elements from each sub-permutation. Hence, the Δ_t operator.

Finally, the s subpartitions are somehow *'mixed'* to form σ'. In general, this mixture operator can be complicated. Yet, as Algorithm 5 preserves the relative order of elements in the sublists, the mixture reduces to a plain shuffle product of the sublists. In terms of labelled structures, this is the labelled product $\star$ of the sub-permutations.

Equation (3.6) only redefines the known class of permutations in a peculiar way. In order to analyze costs, we have to assign each permutation the costs for sorting it. For this, we use the notion of **weighted combinatorial classes**. For a combinatorial class $\mathcal{A}$ and some weight function $w : \mathcal{A} \to \mathbb{R}$, $w(\mathcal{A})$ denotes the weighted class of structures. It consists formally of objects $w(a).a$ for $a \in \mathcal{A}$, i. e. we add to each structure a a "tag" storing its weight $w(a)$. An unweighted class corresponds to weight $w(a) = 1$ for all $a \in \mathcal{A}$.

As with ordinary classes, we assign generating functions to weighted classes. As we deal with labelled structures, we are in the realm of *exponential* generating functions. Class $w(\mathcal{A})$ gets assigned

$$A(z) = \sum_{a \in \mathcal{A}} w(a) \frac{z^{|a|}}{|a|!} = \sum_{n \geqslant 0} \underbrace{\sum_{\substack{a \in \mathcal{A}: \\ |a| = n}} w(a)}_{\widehat{A}_n :=} \frac{z^n}{n!} \, .$$

So, $\widehat{A}_n$ is the total weight of all structures of size n and $A(z)$ is the exponential generating function of the sequence $\widehat{A}_n$. Note that $A(z)$ is at the same time the *ordinary* generating function for sequence $A_n := \frac{\widehat{A}_n}{n!}$. If there are exactly $n!$ structures of size n — as in the case of permutations — this view is handy, since then, A_n is the *average* weight of a structure of size n.

Instead of defining a cost function C explicitly, Hennequin defines the weighted class $C(\mathcal{S})$ recursively along the decomposition of (3.6), cf. [Hen91, Théorème III.1]:

$$\begin{aligned} C(\mathcal{S}) = \quad & T_M \Big[\circ_k \underbrace{\big(\Delta_t\big(C(\mathcal{S})\big) \star \Delta_t(\mathcal{S}) \star \cdots \star \Delta_t(\mathcal{S}) \big)}_{s \text{ times}} \Big] \\ + & T_M \Big[\circ_k \underbrace{\big(\Delta_t(\mathcal{S}) \star \Delta_t\big(C(\mathcal{S})\big) \star \cdots \star \Delta_t(\mathcal{S}) \big)}_{s \text{ times}} \Big] \\ & \vdots \\ + & T_M \Big[\circ_k \underbrace{\big(\Delta_t(\mathcal{S}) \star \cdots \star \Delta_t(\mathcal{S}) \star \Delta_t\big(C(\mathcal{S})\big) \big)}_{s \text{ times}} \Big] \\ + & T_M \big[PC(\mathcal{S}) \big] + R_M \big[C_{SLS}(\mathcal{S}) \big] \; , \end{aligned} \tag{3.7}$$

where $PC(\mathcal{S})$ respectively $C_{SLS}(\mathcal{S})$ are the weighted classes of permutations, where we assign each permutation σ the cost to partition it respectively to sort it with SmallListSort. Note, that the $+$ in eq. (3.6) is the sum of *disjoint* classes and hence plays the role of set union. In eq. (3.7), however, the summands are not disjoint. Hence, we add up cost contributions for permutations from all summands. Equation (3.7) implicitly defines the cost function C recursively: If $|\sigma| \leqslant M$, $C(\sigma)$ is the cost for SmallListSorting σ. Otherwise, the cost is the sum of the partitioning costs and the costs for sorting each of the sub-permutations of σ.

3.5.3 Translation to Generating Functions

Equation (3.7) translates to an equation on generating functions. We define

$$\begin{aligned} S(z) &:= \sum_{\sigma \in \mathcal{S}} \frac{z^{|\sigma|}}{|\sigma|!} = \sum_{n} n! \frac{z^n}{n!} = \frac{1}{1-z} \\ \text{and } C(z) &:= \sum_{\sigma \in \mathcal{S}} C(\sigma) \frac{z^{|\sigma|}}{|\sigma|!} = \sum_{n} \hat{C}_n \frac{z^n}{n!} \\ \text{with } \hat{C}_n &:= \sum_{\sigma \in \mathcal{S}_n} C(\sigma) \end{aligned}$$

and similarly $PC(z) = \sum_{\sigma \in \mathcal{S}} PC(\sigma) \frac{z^{|\sigma|}}{|\sigma|!}$. Now, we can apply the following rules to translate operations on (weighted) combinatorial classes to operations on their exponential generating functions:

- The sum $\mathcal{A} + \mathcal{B}$ becomes the sum of generating functions $A(z) + B(z)$. [FS09, Theorem II.1]
- The labelled product $\mathcal{A} \star \mathcal{B}$ becomes the product of generating functions $A(z)B(z)$. [FS09, Theorem II.1]
- Deletion $\Delta_t(\mathcal{A})$ becomes $A^{(t)}(z)/t!$, with $A^{(t)}$ the tth derivative. [Hen89, Lemme III.3]
- Rooting $\circ_k(\mathcal{A})$ becomes $\underbrace{\int \cdots \int}_{k} k! A(z)\, dz^k$ [Hen89, Lemme III.3].
- The *truncation* and *remainder* operators T_M and R_M can be directly applied to generating functions. For $A(z) = \sum_{n \geqslant 0} a_n z^n$ they are defined by

  $$\begin{aligned} T_M\big[A(z)\big] &:= \sum_{n>M} a_n z^n = A(z) - \sum_{n=0}^{M} a_n z^n , \\ R_M\big[A(z)\big] &:= \sum_{n=0}^{M} a_n z^n . \end{aligned}$$

  It follows quite directly from the definition that we can interchange derivatives and truncation respectively remainders:

  $$O_M\big[A(z)\big]^{(k)} = O_{M-k}\big[A^{(k)}(z)\big] \qquad \text{for } O \in \{T, R\} . \qquad \text{[Hen91, page 40]}$$

Using all those rules, we get the following differential equation out of eq. (3.7):

$$C^{(k)}(z) = T_{M-k}\left[PC^{(k)}(z) + s \cdot k!\, \frac{C^{(t)}(z)}{t!} \left(\frac{S^{(t)}(z)}{t!} \right)^{s-1} \right] + R_{M-k}\big[C_{SLS}{}^{(k)}(z)\big] \tag{3.8}$$

$$\text{with } C_{SLS}(z) = \sum_{\sigma \in \mathcal{S}} C_{SLS}(\sigma) \frac{z^{|\sigma|}}{|\sigma|!} .$$

The differential equation can be rewritten in the form $R(\theta).C(z) = H(z)$ for a function $H(z)$ independent of $C(z)$ and a polynomial R of a *differential operator* θ defined by $\theta.f := (1-z)\frac{d}{dz}f$. Such differential equations can be transformed into a sequence of first order equations by factorizing R. Then, these first order differential equations can be solved explicitly, e. g. by the *integrating factor* method for any fixed t and s. Section 4.2.2.1 on page 62 gives more information of this type of differential equations and also gives an explicit solution for the special case $s = 3$ and $t = 0$.

However, an explicit solution of eq. (3.8) for arbitrary right hand side $H(z)$ is hard to obtain. Instead, Hennequin uses $\mathcal{O}$-transfer lemmas as introduced in [FO90] to obtain asymptotic expansions of $C_n = \hat{C}_n/n!$ directly from $R(\theta)$ and $H(z)$. More precisely, the following steps are taken:

First, Proposition III.4 of [Hen91], gives an explicit solution to $R(\theta).C(z) = H(z)$ for $H(z) = (1-z)^\beta \ln^p(1-z)$. Then, [Hen91]'s Corollaire III.3 says that bounds on $H(z)$ like $H(z) = \mathcal{O}\big((1-z)^\beta \ln^p(1-z)\big)$ can be *transferred* to the solution. Together, this means that we can develop $H(z)$ in terms $(1-z)^\beta \ln^p(1-z)$ and then directly determine the coefficients of $C(z)$ up to a corresponding error term.

The differential equation (3.8) does not determine $C(z)$ uniquely. For example, the solution by integrating factors shows that we do as many integrations as the degree of R. Accordingly, we get the same number of undetermined integration constants. These constants have to be derived from the initial conditions, i. e. the costs for small lists. However, it turns out that these integration constants do not contribute to the leading term of the coefficients of $C(z)$. So, we can directly compute the leading term from the partitioning cost [Hen91, Corollaire III.4]:

$$C_n = [z^n]F(z) + \mathcal{O}(n)$$

for $F(z)$ an arbitrary solution of

$$R(\theta).F(z) = \frac{(1-z)^k PC^{(k)}(z)}{k!} .$$

Such a particular solution to the last equation can then be computed for given $PC(z)$.

3.5.4 Leading Terms for Algorithm 5

From the generating functions derived in the last section, one can compute the expected number of comparison needed by Algorithm 5 on a random permutation of length n. The expected number of comparisons used by Algorithm 5 in one partitioning step is the number of non-pivot elements times the expected costs of an unsuccessful search in a perfect binary search tree of $s-1$ elements, given in eq. (3.5). The coefficient of the leading $n \ln n$ term does not depend on M. Its value for small s and t are listed in Table 2.

With respect to this thesis, the following observation is remarkable: If a random pivot is chosen, Algorithm 5 with $s = 3$ performs asymptotically the same number of comparisons

$\cdot n \ln n + \mathcal{O}(n)$	$t = 0$	$t = 1$	$t = 2$	$t = 3$	$t = 4$	$t = 5$
$s = 2$	2	1.714	1.6216	1.5760	1.5489	1.5309
$s = 3$	2	1.754	1.6740	1.6342	1.6105	1.5947
$s = 4$	1.846	1.642	1.5750	1.5415	1.5216	1.5083
$s = 5$	1.870	1.679	1.6163	1.5848	1.5659	1.5534
$s = 6$	1.839	1.663	1.6047	1.5755	1.5579	1.5463
$s = 7$	1.793	1.631	1.5768	1.5496	1.5333	1.5224

Table 2: Leading terms of the number of comparisons needed by Algorithm 5 to sort a random permutation for small values of s and t. This table is an excerpt from [Hen91, Tableau D.3].

as with $s = 2$. Moreover, if we choose the pivots as the tertiles of a sample of $3t - 2$ elements, Algorithm 5 does significantly *more* comparisons than classic Quicksort with median of $2t - 1$. Note that the sample used for pivot selection by the dual pivot variant is about 50 % larger and still it performs worse! In fact, judging from this table, $s = 3$ seems to be a *extraordinary bad* choice: It contains the maximum in every column.

3.6 Discussion

Quicksort might be the algorithm, we understand best and know most about. Therefore, any reasonably sized summary of previous work on Quicksort is doomed to remain incomplete. For sure, many significant contributions have not been granted the room they deserve — for that I apologize.

The focus of this thesis is on directly implementable algorithms that are useful in practice. I tried my best to present a comprehensive collection of proposed variants of the basic Quicksort algorithm, which aim exactly at improving Quicksort's efficiency in practice. Some of these variants came to fame by being adopted as improvements for program library sorting methods. Others have sunk into oblivion — often legitimately as closer investigation showed them to be unprofitable.

It has long been believed that multi-pivot Quicksort is among these not promising variants. In [Sed75], Sedgewick analyzes Algorithm 7 and finds out that it needs an excessive number of swap operations compared with classic Quicksort. In the early 1990s, Hennequin studied the number of comparisons needed by the parametric multi-pivot Quicksort (Algorithm 5) in his thesis [Hen91]. As shown in Section 3.5.4, choosing two pivots is rather detrimental, there. In light of these results, it is not surprising to see two decades pass without much efforts in this direction.

Then, in 2009, Yaroslavskiy's algorithm (Algorithm 8) appeared out of thin air and turned the world of Java sorting methods upside down. It will be the objective of the following chapters to shed some light on this new algorithm and possible reasons for its success.

4 Average Case Analysis of Dual Pivot Quicksort: Counting Swaps and Comparisons

> “ *People who analyze algorithms have double happiness. First of all they experience the sheer beauty of elegant mathematical patterns that surround elegant computational procedures. Then they receive a practical payoff when their theories make it possible to get other jobs done more quickly and more economically.* ”
>
> — D. E. Knuth in the Foreword of [SF96]

4.1 Setting for the Analysis

4.1.1 The Algorithms

In this Chapter, I will analyze two Quicksort variants that partition the current list into $s = 3$ partitions around two pivots. Sedgewick introduces the implementation given in Algorithm 7 in his PhD thesis [Sed75]. To the knowledge of the author, this is the first implementation of a dual pivot Quicksort in a procedural language. Sedgewick also gives a precise average case analysis of it, which I will reproduce here.

In 2009, Yaroslavskiy proposed a new dual pivot Quicksort implementation on the Java core library mailing list. The discussion is archived at [Jav09]. The original proposal was in the form of a Java program, which I distilled down to Algorithm 8.

The version of Yaroslavskiy's algorithm which was finally accepted for the Oracle's Java 7 runtime library incorporates some variants that turned out beneficial in performance tests. Most notably, the library implementation selects as pivots the *tertiles of a sample of five elements*. Moreover, it features special treatment of duplicate keys: Elements with keys equal to one of the pivots end up in the middle partition. In an additional scan over this middle part, such elements are moved to the left and right end of the middle partition, respectively. Then, they can be excluded from ranges of recursive calls.

The aim of this chapter is to compare *basic* partitioning schemes — the focus is on *understanding* the differences in efficiency, not on building the ultimate Quicksort implementation right away. Therefore Algorithms 7 and 8 do not contain any of the mentioned variants.

Algorithm 7. Dual Pivot Quicksort with SEDGEWICK's partitioning. This algorithm appears as Program 5.1 in [Sed75].

```
DualPivotQuicksortSedgewick($A$, left, right)
    // Sort the array $A$ in index range left, ..., right.
    if $right - left \geqslant 1$
        $i := left$;      $i_1 := left$
        $j := right$;     $j_1 := right$
        $p := A[left]$;   $q := A[right]$
        if $p > q$ then Swap $p$ and $q$ end if
        while true
            $i := i + 1$
            while $A[i] \leqslant q$
                if $i \geqslant j$ then break outer while end if     // pointers have crossed
                if $A[i] < p$
                    $A[i_1] := A[i]; i_1 := i_1 + 1; A[i] := A[i_1]$
                end if
                $i := i + 1$
            end while
            $j := j - 1$
            while $A[j] \geqslant p$
                if $A[j] > q$
                    $A[j_1] := A[j]; j_1 := j_1 - 1; A[j] := A[j_1]$
                end if
                if $i \geqslant j$ then break outer while end if     // pointers have crossed
                $j := j - 1$
            end while
            $A[i_1] := A[j]$; $A[j_1] := A[i]$
            $i_1 := i_1 + 1$;   $j_1 := j_1 - 1$
            $A[i] := A[i_1]$; $A[j] := A[j_1]$
        end while
        $A[i_1] := p$
        $A[j_1] := q$
        DualPivotQuicksortSedgewick($A$,  left  , $i_1 - 1$)
        DualPivotQuicksortSedgewick($A, i_1 + 1, j_1 - 1$)
        DualPivotQuicksortSedgewick($A, j_1 + 1$, right )
    end if
```

Algorithm 8. Dual Pivot Quicksort with YAROSLAVSKIY's partitioning.

```
DualPivotQuicksortYaroslavskiy($A$, $left$, $right$)
    // Sort the array $A$ in index range $left, \dots, right$.
    if $right - left \geqslant 1$
        $p := A[left]$;   $q := A[right]$
        if $p > q$ then Swap $p$ and $q$ end if
        $\ell := left + 1$;   $g := right - 1$;   $k := \ell$
        while $k \leqslant g$
            if $A[k] < p$
                Swap $A[k]$ and $A[\ell]$
                $\ell := \ell + 1$
            else
                if $A[k] \geqslant q$
                    while $A[g] > q$ and $k < g$ do $g := g - 1$ end while
                    Swap $A[k]$ and $A[g]$
                    $g := g - 1$
                    if $A[k] < p$
                        Swap $A[k]$ and $A[\ell]$
                        $\ell := \ell + 1$
                    end if
                end if
            end if
            $k := k + 1$
        end while
        $\ell := \ell - 1$;   $g := g + 1$
        Swap $A[left]$ and $A[\ell]$   // Bring pivots to final position
        Swap $A[right]$ and $A[g]$
        DualPivotQuicksortYaroslavskiy($A$, $left$ , $\ell - 1$)
        DualPivotQuicksortYaroslavskiy($A$, $\ell + 1$, $g - 1$)
        DualPivotQuicksortYaroslavskiy($A$, $g + 1$, $right$)
    end if
```

Resulting from the discussion [Jav09], Yaroslavskiy published a summary document describing the new Quicksort implementation [Yar09]. Note, however, that the analysis in the above document gives merely a rough estimate of the actual expected costs as it is based on overly pessimistic assumptions. In particular, it fails to notice the savings in the number of comparisons Yaroslavskiy achieves over classic Quicksort, see Tables 1 and 3.

A Note on Equal Elements

Even though for the rest of this thesis, we will assume elements to be pairwise distinct, let us for a moment consider the case of equal keys. It is most convenient to argue for the most degenerate case: a list where *all* elements are equal. On such inputs, Algorithm 7 slides into quadratic runtime complexity. The reason is that the comparison in line 8 is always true, so i runs all the way up until it meets j. As a consequence, we get worst case partitioning in every step.

There is a very simple modification that can make up for this without introducing additional costs in the case of distinct keys: We merely have to make the comparisons in lines 8 and 16 *strict*, i. e. $A[i] < q$ and $A[j] > p$ instead of $\leqslant q$ and $\geqslant p$, respectively. Then, the inner loops are never entered and i_1 and j_1 meet in the middle. This results in sublists of relative lengths $\frac{1}{2}$, 0 and $\frac{1}{2}$, which is not perfect but quite good — way better than $0, 0, 1$! In fact, this is very similar to the way Algorithm 1 operates on lists of all equal elements.

Since we will replace Algorithm 7 by the improved variant Algorithm 9 as a result of this chapter, I leave Algorithm 7 unchanged. For Algorithm 9 however, the modification is incorporated.

As mentioned above, the actual Java runtime implementation of Yaroslavskiy's algorithm has special code for dealing with equal elements to avoid quadratic behavior. However, this is in fact not needed! By making the comparison in line 10 of Algorithm 8 non-strict, we achieve that k and g meet in the middle, which results again in relative sublist lengths of $\frac{1}{2}$, 0 and $\frac{1}{2}$. I allowed myself to include this optimization in Algorithm 8 right away as my personal contribution to Yaroslavskiy's partitioning method: the line underneath $>$.

4.1.2 Input Model

All analyses are done in the *random permutation model*, i. e. input sequences are assumed to be random permutations of elements $\{1, \ldots, n\}$, where each permutation occurs with probability $1/n!$. As the algorithms only rely on element comparisons and not on the actual values of elements, we can identify a list element with its *rank* inside the list. Note further, that the random permutation model implies the same behavior as lists of i. i. d. uniformly chosen reals.

The pivots are chosen at fixed positions, namely the first and last elements of the list. Let the smaller one be p, the larger one q. In the random permutation model, this is equivalent to uniformly selecting random pivots since the probability to end up in a certain position is the same for every element and position.

4.1.3 Elementary Operations of Sorting

In this chapter, we analyze the dual pivot Quicksort variants in terms of an abstract cost model. The efficiency of an algorithm is identified with how often an algorithm needs to make use of certain elementary operations. This provides a rough estimate of its quality, which is really a property of the *algorithm*, not its implementation. Hence, it is also absolutely machine-independent.

According to the problem definition in Section 2.2 on page 16, our sorting algorithms can only use the relative order of elements, not their absolute values. The most elementary building block of "determining the relative order" is a **key comparison** of two elements. Hence, the number of such comparisons needed to fully sort a given list of elements is a useful measure for the efficiency of an algorithm.

One reason why Quicksort is often used in practice is that it can be implemented *in-place*, i. e. if the input list is given as a random access array of memory, we can directly sort this list without having to copy the input. All Quicksort implementations studied in this thesis are of this kind. [Knu98, exercise 5.2.2.20] shows that Quicksort is guaranteed to get along with $\mathcal{O}(\log n)$ additional memory, if we apply tail-recursion elimination and avoid sorting the largest sublist first.

An in-place implementation of Quicksort can only work with a constant number of array elements at a time directly, i. e. without reading them from the array. Indeed, the implementations considered here will only read two array elements (in addition to the pivots) — and potentially write them at a different location — before loading the next elements. We will refer to this process of loading two elements and storing them again as one **swap**. The number of such swaps an algorithm does in order to sort a list is a second measure of its efficiency.

Note that in Algorithm 7, the elements are not really exchanged, but written back one position apart from the old position of their swap partner. As the runtime contributions will be roughly the same as for a real exchange, we allow this slightly sloppy definition of a swap.

In Chapter 7, we will use a more detailed measure of efficiency, namely the number of executed *machine instructions* for two specific machines. There, we will show that the leading term is a linear combination of the number of swaps and the number of comparisons. Hence, even for this much more detailed measure, swaps and comparisons are the only elementary operations yielding an asymptotically dominant contribution to the costs.

4.2 Recurrence Relation

Note that all Quicksort variants in this thesis fulfill the following property:

Property 1: Every key comparison involves a pivot element of the current partitioning step.

In [Hen89], Hennequin shows that Property 1 is a sufficient criterion for *preserving randomness* in subfiles: If the whole array is a (uniformly chosen) random permutation of its elements, so are the subproblems Quicksort is recursively invoked on. This allows us to set up a recurrence relation for the expected costs, as it ensures that all partitioning steps of a subarray of size k have the same expected costs as the initial partitioning step for a random permutation of size k.

The expected costs C_n for sorting a random permutation of length n by any dual pivot Quicksort with Property 1 satisfy the following recurrence relation:

$$\begin{aligned}
C_0 &= 0 \\
C_1 &= 0 \\
C_n &= \sum_{1 \leqslant p < q \leqslant n} \Pr[\text{pivots } (p,q)] \cdot (\text{partitioning costs} + \text{recursive costs}) \qquad (4.1) \\
&= \sum_{1 \leqslant p < q \leqslant n} 1/\tbinom{n}{2} \cdot \bigl(\text{partitioning costs} + C_{p-1} + C_{q-p-1} + C_{n-q}\bigr) \\
&= pc_n \quad + \quad 1/\tbinom{n}{2} \sum_{1 \leqslant p < q \leqslant n} \bigl(C_{p-1} + C_{q-p-1} + C_{n-q}\bigr) \,, \qquad (n \geqslant 2)
\end{aligned}$$

where pc_n is the expected partitioning cost for a list of length n. By inserting appropriate toll functions pc_n, we will later use this recurrence to compute the expected number of swaps and comparisons.

In the following two subsections, I give two independent derivations of the closed form of C_n. Section 4.2.1 uses elementary rearrangements — in particular clever forward differences of the sequence C_n — to find a different recursive, but *telescoping* characterization of C_n. Such a representation is then immediately written as an explicit sum. In the end, this allows us to express C_n directly in terms of pc_n.

The second derivation presented in Section 4.2.2 is based on the symbolic method (see [FS09, Part A]): We set up a functional equation for the *generating function* $C(z)$ of C_n, solve this equation to obtain a closed for $C(z)$ and then determine C_n as the coefficients of $C(z)$.

Whereas the elementary derivation is more self-contained and essentially doable with high-school math, the generating function approach requires us to deal with tractable, yet non-trivial differential equations. On the other hand, Section 4.2.1 contains some "guess-and-prove" parts, which are hard to generalize. Here, Hennequin's approach shines: The symbolic description of the corresponding generating functions already includes an arbitrary number of pivots as well as choosing these from a larger sample. (These variants are introduced in Section 3.4.1 "Choice of the Pivot" and Section 3.4.4 "Multi-Pivot Quicksort")

4.2.1 Elementary Solution

The solution presented in this section is a generalization of SEDGEWICK's derivation of the expected number of swaps for Algorithm 7 in [Sed75, p. 156ff]. Although the computations in this section allow to actually *derive* the closed form — in contrast to only *verifying* a 'guessed' solution — some steps follow from 'magic' insight. The derivation in Section 4.2.2 will appear more directed to the exercised generatingfunctionologist. On the other hand, the math is more involved.

The first step is to use the symmetry in the sum of recursive costs in eq. (4.1).[14]

$$\begin{aligned}
&\sum_{1\leqslant p<q\leqslant n} \bigl(C_{p-1} + C_{q-p-1} + C_{n-q}\bigr) \\
&= \sum_{1\leqslant p<q\leqslant n} C_{p-1} \quad + \sum_{1\leqslant p<q\leqslant n} C_{q-p-1} \quad + \sum_{1\leqslant p<q\leqslant n} C_{n-q} \\
&= \sum_{p=1}^{n-1} C_{p-1} \sum_{q=p+1}^{n} 1 \quad + \sum_{k=0}^{n-2} C_k \sum_{p=1}^{n-1-k} 1 \quad + \sum_{q=2}^{n} C_{n-q} \sum_{p=1}^{q-1} 1 \\
&= \sum_{p=1}^{n-1} C_{p-1} \; (n-p) \quad + \sum_{k=0}^{n-2} C_k(n-1-k) \quad + \sum_{q=2}^{n} C_{n-q} \; (q-1) \\
&= \sum_{k=0}^{n-2} C_k \; (n-k-1) \quad + \sum_{k=0}^{n-2} C_k(n-k-1) \quad + \sum_{k=0}^{n-2} C_k(n-k-1) \\
&= 3\sum_{k=0}^{n-2} (n-k-1)C_k \;.
\end{aligned}$$

So, our recurrence to solve is

$$\begin{aligned}
C_0 &= C_1 = 0 \\
C_n &= pc_n + \tfrac{6}{n(n-1)} \sum_{k=0}^{n-2} (n-k-1)C_k \qquad \text{for } n \geqslant 2 \;.
\end{aligned}$$

We first consider $D_n := \binom{n+1}{2}C_{n+1} - \binom{n}{2}C_n$ to get rid of the factor in the sum:

$$\begin{aligned}
D_n &= \overbrace{\tbinom{n+1}{2}pc_{n+1} - \tbinom{n}{2}pc_n}^{d(n):=} \qquad\qquad (n \geqslant 3) \\
&\quad + \tfrac{(n+1)n}{2}\tfrac{6}{(n+1)n} \sum_{k=0}^{n-1} (n-k)C_k - \tfrac{n(n-1)}{2}\tfrac{6}{n(n-1)} \sum_{k=0}^{n-2} (n-k-1)C_k \\
&= d(n) + 3\sum_{k=0}^{n-1} C_k \;.
\end{aligned}$$

[14] Please note: I tried to give all algebraic manipulations in great detail in this section, in the hope that this will allow the reader to follow the derivation without additional scratch paper. The veteran mathematician bored by sissy-style calculations is advised to skip every other line or so of the formulæ as to keep the reading demanding enough.

The remaining full history recurrence is eliminated by taking ordinary differences

$$\begin{aligned} E_n &:= D_{n+1} - D_n \\ &= d(n+1) - d(n) + 3C_n \,. \qquad (n \geqslant 3) \end{aligned}$$

Towards a telescoping recurrence, we consider yet another quantity

$$F_n := C_n - \tfrac{n-4}{n} \cdot C_{n-1} \,,$$

and compute

$$\begin{aligned} F_{n+2} - F_{n+1} &= C_{n+2} - \tfrac{n-2}{n+2} C_{n+1} - \left(C_{n+1} - \tfrac{n-3}{n+1} C_n \right) \\ &= C_{n+2} - \tfrac{2n}{n+2} C_{n+1} + \tfrac{n-3}{n+1} C_n \,. \end{aligned}$$

The expression on the right hand side is not quite appealing. However, by expanding the definition of E_n, we find

$$\begin{aligned} (E_n - 3C_n) \,/\, \tbinom{n+2}{2} &= (D_{n+1} - D_n - 3C_n) \,/\, \tbinom{n+2}{2} \\ &= \Bigg(\tbinom{n+2}{2} C_{n+2} - \tbinom{n+1}{2} C_{n+1} \\ &\qquad - \Big(\tbinom{n+1}{2} C_{n+1} - \tbinom{n}{2} C_n \Big) - 3C_n \Bigg) \Big/ \tbinom{n+2}{2} \\ &= C_{n+2} - 2\tbinom{n+1}{2} / \tbinom{n+2}{2} \cdot C_{n+1} + \left(\tbinom{n}{2} - 3 \right) / \tbinom{n+2}{2} \cdot C_n \\ &= C_{n+2} - \tfrac{2n}{n+2} C_{n+1} + \tfrac{\frac{1}{2} n(n-1) - 3}{\frac{1}{2}(n+2)(n-1)} C_n \\ &= C_{n+2} - \tfrac{2n}{n+2} C_{n+1} + \tfrac{\frac{1}{2}(n-3)(n+2)}{\frac{1}{2}(n+2)(n-1)} C_n \\ &= C_{n+2} - \tfrac{2n}{n+2} C_{n+1} + \tfrac{n-3}{n+1} C_n \,. \end{aligned}$$

Hence, we can equate these two terms to get

$$F_{n+2} - F_{n+1} = (E_n - 3C_n) \,/\, \tbinom{n+2}{2} = \underbrace{\left(d(n+1) - d(n) \right) \,/\, \tbinom{n+2}{2}}_{f(n) :=} \,. \qquad (n \geqslant 3)$$

This last equation is now amenable to simple iteration:

$$F_n = \underbrace{\sum_{i=5}^{n} f(i-2) + F_4}_{g(n)} \,. \qquad (n \geqslant 5)$$

Plugging in the definition of $F_n = C_n - \frac{n-4}{n} \cdot C_{n-1}$ yields

$$C_n = \tfrac{n-4}{n} \cdot C_{n-1} \; + g(n) \,.$$

Multiplying by $\binom{n}{4}$ and using $\binom{n}{4} \cdot \frac{n-4}{n} = \binom{n-1}{4}$ gives a telescoping recurrence for $G_n := \binom{n}{4} C_n$:

$$\begin{aligned} G_n &= G_{n-1} + \tbinom{n}{4} g(n) \\ &= \sum_{i=5}^{n} \tbinom{i}{4} g(i) + G_4 \\ &= \sum_{i=1}^{n} \tbinom{i}{4} g(i) - \tbinom{4}{4} F_4 + G_4 \\ &= \sum_{i=1}^{n} \tbinom{i}{4} g(i) - \underbrace{\left(C_4 - \tfrac{4-4}{4} C_3\right) + \tbinom{4}{4} C_4}_{=0} \end{aligned}$$

Finally, we arrive at an explicit formula for C_n, which is valid for $n \geqslant 4$:

$$\begin{aligned} C_n = G_n \,/\, \tbinom{n}{4} &= \frac{1}{\binom{n}{4}} \cdot \sum_{i=1}^{n} \tbinom{i}{4} g(i) \\ &= \frac{1}{\binom{n}{4}} \cdot \sum_{i=1}^{n} \tbinom{i}{4} \left(\sum_{j=3}^{i-2} f(j) + F_4 \right) \\ &= \frac{1}{\binom{n}{4}} \cdot \sum_{i=1}^{n} \tbinom{i}{4} \left(F_4 + \sum_{j=3}^{i-2} \frac{d(j+1) - d(j)}{\binom{j+2}{2}} \right) \\ &= \frac{1}{\binom{n}{4}} F_4 \cdot \sum_{i=1}^{n} \tbinom{i}{4} \\ &\quad + \frac{1}{\binom{n}{4}} \cdot \sum_{i=1}^{n} \tbinom{i}{4} \sum_{j=3}^{i-2} \frac{\binom{j+2}{2} pc_{j+2} - 2\binom{j+1}{2} pc_{j+1} + \binom{j}{2} pc_j}{\binom{j+2}{2}} \\ &= F_4 \cdot \tbinom{n+1}{5} \,/\, \tbinom{n}{4} \\ &\quad + \frac{1}{\binom{n}{4}} \sum_{i=1}^{n} \tbinom{i}{4} \sum_{j=3}^{i-2} \left(pc_{j+2} - \tfrac{2j}{j+2} pc_{j+1} + \frac{\binom{j}{2}}{\binom{j+2}{2}} pc_j \right) \end{aligned}$$

Using $F_4 = C_4 = pc_4 + \frac{1}{2} pc_2$, this simplifies to

$$C_n = \frac{1}{\binom{n}{4}} \sum_{i=1}^{n} \tbinom{i}{4} \sum_{j=3}^{i-2} \left(pc_{j+2} - \tfrac{2j}{j+2} pc_{j+1} + \frac{\binom{j}{2}}{\binom{j+2}{2}} pc_j \right) + \tfrac{n+1}{5} \left(pc_4 + \tfrac{1}{2} pc_2 \right) . \tag{4.2}$$

The term for C_n in eq. (4.2) is closed in the sense that it only depends on the partitioning costs pc_n, even though it still involves a non-trivial double sum over binomials. Luckily, such sums are often amenable to computer algebra, see [PWZ96]. Another nice property is that **C_n is linear in pc_n**, so if pc_n is a linear combination, we can compute the above sum of each of the summands separately.

How to proceed manually is illustrated in the next section. There, I derive the total costs C_n for a parametric linear partitioning cost definition (see eq. (4.3)). The resulting closed form will be general enough for all partitioning costs we encounter for Algorithm 8. Similar to this computation, one can determine the total costs for some non-linear partitioning costs pc_n, as well.

4.2.1.1 Linear Partitioning Costs

For the toll functions pc_n we will encounter, eq. (4.2) can be written in a rather succinct form in terms of harmonic numbers $\mathcal{H}_n$. Let us set

$$pc_n = \begin{cases} 0 & n < 2 \\ d & n = 2 \\ a(n+1)+b & n > 2 \end{cases} . \tag{4.3}$$

The reason for this somewhat peculiar definition is that typical partitioning costs are essentially linear in n, but behave differently for lists of size $\leqslant 2$. For the inner sum in eq. (4.2), we only use the values pc_i with $i \geqslant 3$. By partial fraction decomposition, we obtain for the summands

$$pc_{j+2} - \tfrac{2j}{j+2} pc_{j+1} + \frac{\binom{j}{2}}{\binom{j+2}{2}} pc_j \quad = \quad \frac{2b}{j+1} + \frac{6a-2b}{j+2} .$$

This allows to split the inner sum and represent it in terms of harmonic numbers:

$$\begin{aligned}
C_n &= \frac{1}{\binom{n}{4}} \sum_{i=1}^{n} \binom{i}{4} \left(2b \sum_{j=3}^{i-2} \tfrac{1}{j+1} + (6a-2b) \sum_{j=3}^{i-2} \tfrac{1}{j+2} \right) + \tfrac{n+1}{5} \left(5a + b + \tfrac{1}{2} d\right) \\
&= \frac{1}{\binom{n}{4}} \sum_{i=1}^{n} \binom{i}{4} \left(2b \sum_{j=4}^{i-1} \tfrac{1}{j} + (6a-2b) \sum_{j=5}^{i} \tfrac{1}{j} \right) + \tfrac{n+1}{5} \left(5a + b + \tfrac{1}{2} d\right) \\
&= \frac{1}{\binom{n}{4}} \sum_{i=1}^{n} \binom{i}{4} \Big(2b(\mathcal{H}_{i-1} - \mathcal{H}_3) + (6a-2b)(\mathcal{H}_i - \mathcal{H}_4) \Big) + \tfrac{n+1}{5} \left(5a + b + \tfrac{1}{2} d\right) \\
&= \frac{1}{\binom{n}{4}} \sum_{i=1}^{n} \binom{i}{4} \Big(6a\,\mathcal{H}_i - 2b(\tfrac{1}{i} + \mathcal{H}_3) - (6a-2b)\mathcal{H}_4 \Big) + \tfrac{n+1}{5} \left(5a + b + \tfrac{1}{2} d\right) \\
&= \frac{1}{\binom{n}{4}} \sum_{i=1}^{n} \binom{i}{4} \Big(6a\,\mathcal{H}_i - 2b\tfrac{1}{i} - 2b\tfrac{11}{6} - (6a-2b)\tfrac{25}{12} \Big) + \tfrac{n+1}{5} \left(5a + b + \tfrac{1}{2} d\right) \\
&= \frac{1}{\binom{n}{4}} \sum_{i=1}^{n} \binom{i}{4} \Big(6a\,\mathcal{H}_i - 2b\tfrac{1}{i} + \tfrac{1}{2}(b - 25a) \Big) + \tfrac{n+1}{5} \left(5a + b + \tfrac{1}{2} d\right) \\
&= \frac{6a}{\binom{n}{4}} \sum_{i=1}^{n} \binom{i}{4} \mathcal{H}_i - \frac{2b}{\binom{n}{4}} \sum_{i=1}^{n} \tfrac{1}{i} \binom{i}{4} + \frac{\tfrac{1}{2}(b - 25a)}{\binom{n}{4}} \sum_{i=1}^{n} \binom{i}{4} + \tfrac{n+1}{5} \left(5a + b + \tfrac{1}{2} d\right) \\
&= \frac{6a}{\binom{n}{4}} \sum_{i=1}^{n} \binom{i}{4} \mathcal{H}_i - \frac{2b}{\binom{n}{4}} \sum_{i=1}^{n} \tfrac{1}{4} \binom{i-1}{3} + \frac{\tfrac{1}{2}(b - 25a)}{\binom{n}{4}} \sum_{i=1}^{n} \binom{i}{4} + \tfrac{n+1}{5} \left(5a + b + \tfrac{1}{2} d\right)
\end{aligned}$$

The first sum is an instance of eq. ($\Sigma i\mathcal{H}_i$) on page 15, which says

$$\sum_{0\leqslant k<n} \tbinom{k}{m}\mathcal{H}_k = \tbinom{n}{m+1}\bigl(\mathcal{H}_n - \tfrac{1}{m+1}\bigr) \qquad \text{for integer } m \geqslant 0\ .$$

The other two sums are plain summations in the upper index (equation (5.10) of [GKP94, page 160]):

$$\sum_{0\leqslant k\leqslant n} \tbinom{k}{m} = \tbinom{n+1}{m+1} \qquad \text{for integers } m, n \geqslant 0\ .$$

Using these, we finally get

$$\begin{aligned} C_n &= \frac{6a}{\binom{n}{4}}\tbinom{n+1}{5}\bigl(\mathcal{H}_{n+1} - \tfrac{1}{5}\bigr) - \frac{b}{2\binom{n}{4}}\tbinom{n}{4} + \frac{\frac{1}{2}(b-25a)}{\binom{n}{4}}\tbinom{n+1}{5} + \tfrac{n+1}{5}\bigl(5a + b + \tfrac{1}{2}d\bigr) \\ &= \tfrac{6}{5}a(n+1)(\mathcal{H}_{n+1} - \tfrac{1}{5}) - \tfrac{1}{2}b + (\tfrac{1}{10}b - \tfrac{25}{10}a)(n+1) + \tfrac{1}{10}(n+1)\,(10a + 2b + d) \\ &= \tfrac{6}{5}a(n+1)\mathcal{H}_{n+1} \ +\ \bigl(-\tfrac{6}{25}a - \tfrac{3}{2}a + \tfrac{3}{10}b + \tfrac{1}{10}d\bigr)\,(n+1) \ -\ \tfrac{1}{2}b \\ &= \tfrac{6}{5}a(n+1)\mathcal{H}_{n+1} \ +\ \bigl(-\tfrac{87}{50}a + \tfrac{3}{10}b + \tfrac{1}{10}d\bigr)\,(n+1) \ -\ \tfrac{1}{2}b\ . \end{aligned} \tag{4.4}$$

4.2.2 Generating Function Solution

I assume basic familiarity with generating functions in this section. A full introduction is beyond the scope of this thesis and excellent textbooks are available [GKP94, Chapter 7], [SF96] and [Wil06]. A thorough treatment of the symbolic method appears in [FS09], which also gives methods to obtain asymptotic approximations from generating functions.

In Section 3.5 on page 43, I summarized Hennequin's analysis of his parametric Quicksort implementation: In the most general form it features partitioning around $s-1$ pivots, chosen equidistantly from a sample of $k = s\cdot t + (s-1)$, and special treatment of small sublists. Hennequin derives a differential equation given in eq. (3.8) on page 48 for the (ordinary) generating function

$$C(z) = \sum_{n\geqslant 0} C_n z^n$$

of the expected costs C_n of sorting a random permutation of size n. On page 40 of [Hen91], Hennequin shows that eq. (3.8) is equivalent to the following recurrence relation on the coefficients C_n of $C(z)$:

$$C_n = \begin{cases} pc_n + \displaystyle\sum_{n_1+\cdots+n_s=n-s+1} \frac{\binom{n_1}{t}\cdots\binom{n_s}{t}}{\binom{n}{k}}(C_{n_1} + \cdots + C_{n_s}) & \text{if } n > M \\ SLS_n & \text{if } n \leqslant M \end{cases} \qquad \text{[Hen91, (III.11)].}$$

For the parameter choices

$$s = 3, \qquad M = 2, \qquad t = 0 \quad (\rightsquigarrow k = 2)\,,$$

corresponding to the basic dual pivot Quicksort implementations we study in this chapter, this recurrence reduces to eq. (4.1). Hence, we can use Hennequin's approach to solve eq. (4.1). Likewise, for $s = 3$, $M = 2$ and $t = 0$, the differential equation (3.8) simplifies to

$$C''(z) = T_0 \left[PC''(z) + 3 \cdot 2 \cdot C(z) \left(\tfrac{1}{1-z}\right)^2 \right] + R_0 \left[C_{SLS}''(z) \right] . \tag{4.5}$$

Recall that T_0 and R_0 are the *truncation* and *remainder* operators (of order 0), respectively. They are defined for general order in Section 3.5.3. For order 0, they simplify to $T_0\left[G(z)\right] = G(z) - G(0)$ and $R_0\left[G(z)\right] = G(0)$. As T_0 is a linear operator, eq. (4.5) becomes

$$\begin{aligned} C''(z) &= \Big(PC''(z) - PC''(0) \Big) + \Big(6 \frac{C(z)}{(1-z)^2} - 6\,C(0) \Big) + C_{SLS}''(0) \\ &= 6 \frac{C(z)}{(1-z)^2} \quad + \quad PC''(z) \quad + \quad \underbrace{C_{SLS}''(0) - PC''(0) - 6\,C(0)}_{Q:=} . \end{aligned}$$

Multiplying by $(1-z)^2$ and rearranging yields

$$(1-z)^2 C''(z) - 6C(z) = (1-z)^2 \big(PC''(z) + Q \big) . \tag{4.6}$$

We observe the correspondence between the order of derivatives and exponents of $(1-z)$. Such differential equations are known as *Cauchy-Euler equations* (also just *Euler equations*) or *equidimensional equations* in the literature, see e. g. [Inc27, Section 6.3]. They allow an explicit solution using the *method of linear operator* by transforming the higher-order differential equation to a sequence of first-order equations.

We introduce the differential operator θ with $\theta.f(z) = (1-z)f'(z)$. Using $\theta(\theta+1).f(z) = (1-z)^2 f''(z)$ we can write eq. (4.6) as

$$\big(\theta(\theta+1) - 6 \big).C(z) = (1-z)^2 \big(PC''(z) + Q \big) .$$

Factorizing $\theta(\theta+1) - 6$ yields

$$(\theta - 2)(\theta + 3).C(z) = (1-z)^2 \big(PC''(z) + Q \big) \tag{4.7}$$

$$\text{with } Q = C_{SLS}''(0) - PC''(0) - 6\,C(0) .$$

Note that $C(0) = C_0 = 0$ by eq. (4.1) and $G''(0) = 2g_2$, so we can actually compute Q rather easily as

$$Q = 2 \left(SLS_2 - pc_2 \right) \tag{4.8}$$

4.2.2.1 Solution to the Differential Equation

If we abbreviate $D(z) := (\theta + 3).C(z)$ in eq. (4.7), we obtain a first order differential equation for $D(z)$:

$$\begin{aligned} & (\theta - 2).D(z) = (1-z)^2 \big(PC''(z) + Q \big) \\ \Longleftrightarrow \qquad & D'(z) - \frac{2}{1-z} D(z) = (1-z) \ \big(PC''(z) + Q \big) \end{aligned}$$

This differential equation can be solved by multiplication with an *integrating factor* $M(z) = e^{\int -\frac{2}{1-z}} = (1-z)^2$:

$$\begin{aligned}
& & M(z)D'(z) - M(z)\frac{2}{1-z}D(z) &= M(z)(1-z)\big(PC''(z)+Q\big) \\
&\iff & M(z)D'(z) + \underbrace{\big(-2(1-z)\big)}_{=M'(z)} D(z) &= (1-z)^3\big(PC''(z)+Q\big) \\
&\iff & \tfrac{d}{dz}\big(M(z)\cdot D(z)\big) &= (1-z)^3\big(PC''(z)+Q\big) \\
&\iff & M(z)\cdot D(z) &= \int (1-z)^3 PC''(z)\,dz + \int (1-z)^3 Q\,dz \\
&\iff & D(z) &= \frac{\int (1-z)^3 PC''(z)\,dz}{(1-z)^2} \;-\; \tfrac{1}{4}Q(1-z)^2 \,.
\end{aligned}$$

Without fixing the partitioning cost, this is the closest form for $D(z)$ we can hope for. All partitioning costs we encounter in the analysis will lead to functions $PC(z)$ for which the antiderivative is easily computed.

To continue, we use the definition of $D(z)$. It gives us a first order differential equation for $C(z)$:

$$\begin{aligned}
(\theta+3).C(z) &= D(z) \\
\iff\; C'(z) + \frac{3}{1-z}C(z) &= \frac{D(z)}{1-z}
\end{aligned}$$

Multiplying with $M(z) = e^{\int \frac{3}{1-z}} = -(1-z)^{-3}$ allows for the same trick as above:

$$\begin{aligned}
& & \tfrac{d}{dz}\big(M(z)\cdot C(z)\big) &= M(z)\cdot\frac{D(z)}{1-z} \\
&\iff & M(z)\cdot C(z) &= \int -\frac{D(z)}{(1-z)^4}\,dz \\
&\iff & C(z) &= (1-z)^3 \int \frac{D(z)}{(1-z)^4}dz \\
& & &= (1-z)^3 \int \frac{\int (1-z)^3 PC''(z)\,dz}{(1-z)^6} - \tfrac{1}{4}Q\frac{1}{(1-z)^2}\,dz \qquad (4.9) \\
& & &= (1-z)^3 \Big(\int \frac{\int (1-z)^3 PC''(z)\,dz}{(1-z)^6}\,dz \;-\; \tfrac{1}{4}Q\frac{1}{1-z}\Big) \\
& & &= (1-z)^3 \int \frac{\int (1-z)^3 PC''(z)\,dz}{(1-z)^6}\,dz \;-\; \tfrac{1}{4}Q(1-z)^2
\end{aligned}$$

4.2.2.2 Linear Partitioning Costs

For Hennequin's approach, we have to define two kinds of primitive costs: Those for an ordinary Quicksort partitioning step and those for SmallListSort. Then, the overall costs can be computed as the coefficients of $C(z)$.

As Section 4.3 will show, the costs pc_n for an ordinary partitioning step are linear for most cost measures, so we set[15]

$$pc_n := a(n+1) \ + \ b \ .$$

Although, we do not explicitly use a procedure SmallListSort, our algorithms treat lists of size $\leqslant M = 2$ differently. In accordance with $C_0 = C_1 = 0$—as given in (4.1)—a one- or zero-element list is already sorted and requires no additional work. Hence $SLS_0 = SLS_1 = 0$. For two-element lists, we choose the symbolic constant $SLS_2 = d$, such that the cost definitions match the definition of pc_n in eq. (4.3) used for Section 4.2.1.

Now that we have fixed our primitive costs, we can compute $C(z)$ from (4.9). Basically, we only have to insert into known terms. However, I will do the calculations in a detailed and easily digestible form. We begin with a warm-up:

$$Q \underset{(4.8)}{=} 2\left(SLS_2 - pc_2\right) = 2d - 6a - 2b \ .$$

Next, we need the ordinary generating function for pc_n:

$$\begin{aligned}
PC(z) &:= \sum_{n\geqslant 0} pc_n z^n \\
&= a\sum_{n\geqslant 0}(n+1)z^n + b\sum_{n\geqslant 0} z^n \\
&= a\sum_{n\geqslant 1} \quad n\,z^{n-1} \quad + b\sum_{n\geqslant 0} z^n \\
&= a\tfrac{d}{dz}\tfrac{1}{1-z} + b\tfrac{1}{1-z} \\
&= \frac{a}{(1-z)^2} + \frac{b}{1-z} \ .
\end{aligned}$$

With these preparations done, we are ready for some calculus exercises: We tackle the double integral in eq. (4.9):

$$\begin{aligned}
\int \frac{\int (1-z)^3 PC''(z)\,dz}{(1-z)^6}\,dz &= \int (1-z)^{-6}\int (1-z)^3\Big(\frac{6a}{(1-z)^4} + \frac{2b}{(1-z)^3}\Big)\,dz\,dz \\
&= \int (1-z)^{-6}\Big(\int \frac{6a}{1-z}\,dz + \int 2b\,dz\Big)\,dz \\
&= \int (1-z)^{-6}\Big(-6a\ln(1-z) - 2b(1-z) + c_1\Big)\,dz \qquad (4.10) \\
&= -6a\int \frac{\ln(1-z)}{(1-z)^6}\,dz - 2b\int \frac{1}{(1-z)^5}\,dz + \int \frac{c_1}{(1-z)^6}\,dz \\
&= -\tfrac{6}{5}a\frac{\ln(1-z)+\frac{1}{5}}{(1-z)^5} - \tfrac{1}{2}b(1-z)^{-4} + \tfrac{1}{5}c_1(1-z)^{-5} + c_2 \ .
\end{aligned}$$

[15] Of course, we could also choose $pc_n := an + b$, but $a(n+1)+b$ will yield a nicer term for the generating function; see below.

We introduced two integration constants c_1 and c_2, which we need to determine from the initial conditions. The first integral in the last line requires integration by parts, so it might be worth zooming in:

$$\begin{aligned}\int \frac{\ln(1-z)}{(1-z)^6}\,dz \underset{z=1-x}{=} & \int \underbrace{\ln(x)}_{u}\underbrace{(-x^{-6})}_{v'}\,dx \\ [\textstyle\int uv' = uv - \int u'v] = \;& \ln(x)\big(\tfrac{1}{5}x^{-5}\big) - \int \tfrac{1}{x}\big(\tfrac{1}{5}x^{-5}\big)\,dx \\ = \;& \tfrac{1}{5}\frac{\ln x}{x^5} - \tfrac{1}{5}\big(-\tfrac{1}{5}x^{-5}\big) \\ \underset{x=1-z}{=} & \frac{\tfrac{1}{5}\ln(1-z) + \tfrac{1}{25}}{(1-z)^5}\,.\end{aligned}$$

Inserting our hard-earned integral (4.10) into eq. (4.9) yields

$$\begin{aligned}C(z) &= (1-z)^3\left(-\tfrac{6}{5}a\frac{\ln(1-z)+\frac{1}{5}}{(1-z)^5} - \tfrac{1}{2}b(1-z)^{-4} + \tfrac{1}{5}c_1(1-z)^{-5} + c_2\right) \;-\; \tfrac{1}{4}Q(1-z)^2 \\ &= \tfrac{6}{5}a\frac{\ln(\frac{1}{1-z})}{(1-z)^2} \;+\; \big(\tfrac{1}{5}c_1 - \tfrac{6}{25}a\big)\frac{1}{(1-z)^2} \;-\; \tfrac{1}{2}b\frac{1}{1-z} \;+\; c_2(1-z)^3 \;-\; \tfrac{1}{4}Q(1-z)^2\,.\end{aligned}$$

Finally, we have the closed for of the generating function $C(z)$ of the average costs for dual pivot Quicksort with linear partitioning costs. From this generating function, we can now obtain the solution to recurrence eq. (4.1) by taking coefficients: $C_n = [z^n]C(z)$. The series expansion of the first summand can be found using equation (7.43) of [GKP94, page 351]:

$$\frac{1}{(1-z)^{m+1}}\ln\frac{1}{1-z} = \sum_{n\geqslant 0}(\mathcal{H}_{m+n} - \mathcal{H}_m)\binom{m+n}{n}z^n\,.$$

The last two summands of $C(z)$ form a polynomial of degree three, so for C_n with $n \geqslant 4$, we can safely ignore them and find

$$\begin{aligned}C_n &= \tfrac{6}{5}a\,(\mathcal{H}_{n+1} - \mathcal{H}_1)\binom{n+1}{n} + \big(\tfrac{1}{5}c_1 - \tfrac{6}{25}a\big)(n+1) - \tfrac{1}{2}b \qquad (n \geqslant 4) \\ &= \tfrac{6}{5}a\mathcal{H}_{n+1}(n+1) + \big(\tfrac{1}{5}c_1 - \tfrac{6}{25}a - \tfrac{6}{5}a\big)(n+1) - \tfrac{1}{2}b \\ &= \tfrac{6}{5}a(n+1)\mathcal{H}_{n+1} + \big(\tfrac{1}{5}c_1 - \tfrac{36}{25}a\big)(n+1) - \tfrac{1}{2}b\,. \qquad (4.11)\end{aligned}$$

For determining the integration constants c_1 and c_2, we use the initial conditions

$$\begin{aligned}0 = C_0 = [z^0]C(z) = C(0) &= \tfrac{1}{5}c_1 - \tfrac{6}{25}a - \tfrac{1}{2}b + c_2 - \tfrac{1}{4}Q \\ 0 = C_1 = [z^1]C(z) = C'(0) &= \tfrac{18}{25}a + \tfrac{2}{5}c_1 - \tfrac{1}{2}b - 3c_2 + \tfrac{1}{2}Q\,.\end{aligned}$$

Solving the linear system for c_1 and c_2 gives:

$$\begin{aligned}c_1 &= -\tfrac{3}{2}a + \tfrac{3}{2}b + \tfrac{1}{2}d\,, \\ c_2 &= -\tfrac{24}{25}a - \tfrac{3}{10}b + \tfrac{2}{5}d\,.\end{aligned}$$

Inserting into eq. (4.11), we finally find our explicit formula for the total cost of dual pivot Quicksort with linear partitioning costs:

$$C_n = \tfrac{6}{5}a(n+1)\mathcal{H}_{n+1} + \big(-\tfrac{87}{50}a + \tfrac{3}{10}b + \tfrac{1}{10}d\big)(n+1) - \tfrac{1}{2}b \tag{4.12}$$

Of course, the result is exactly the same as eq. (4.4) which we found in Section 4.2.1.

4.3 Partitioning Costs

In this section, we analyze the expected number of swaps and comparisons used in the *first partitioning step* on a random permutation of $\{1, \dots, n\}$ by Algorithms 7 and 8. Inserting these as partitioning costs pc_n in eq. (4.2) yields the total number of swaps respectively comparisons for sorting the permutation.

In the following sections, we will always assume $n \geqslant 3$ if not otherwise stated. For $n = 0, 1$, Algorithms 7 and 8 do not execute the partitioning step at all. For $n = 2$, some execution frequencies behave differently, so we treat those as special cases. Of course, contributions for $n = 2$ are trivially determined by a sharp look at the corresponding algorithm.

The following sections introduce some terms and notations, which are used in Sections 4.3.3 and 4.3.4 to compute the expected number of swaps and comparisons.

4.3.1 Notations

The first call to Quicksort takes the form DualPivotQuicksortSedgewick$(A, 1, n)$ respectively DualPivotQuicksortYaroslavskiy$(A, 1, n)$. So, for analyzing the first partitioning step, we can identify *left* $= 1$ and *right* $= n$.

Algorithms 7 and 8 use swaps and key comparisons at several locations in the code. It will pay off to determine the expected execution counts *separately* for all these locations. Assume we scan the code for key comparison instructions and number them consecutively. Then, each comparison done when running the algorithm can be traced back to one of these locations. Now, denote by $\mathtt{c}_i$ the **frequency of the ith comparison location** in the first partitioning step, i. e. how often the comparison location with number i is reached during the first partitioning step.

Its expected value on a random permutation of size n is written as $\mathbb{E}_n\, \mathtt{c}_i$. If n is clear from the context, I simply write $\mathbb{E}\, \mathtt{c}_i$. As an intermediary step, I will often compute $\mathbb{E}_n\, [\mathtt{c}_i \mid p, q]$, the **conditional expected value of $\mathtt{c}_i$ given that the random permutation induces pivots p and q**. For swaps, I use similar definitions based on $\mathtt{s}_i$, the **frequency of the ith swap location**.

In the end, we are interested in the total number of comparisons and swaps, not only counting the *first* partitioning step. I will use $\mathtt{C}_i(n)$ and $\mathtt{S}_i(n)$ to denote the **expected total number** of comparison respectively swaps **from the ith location for a random permutation of size n**. Unless the dependence on n needs explicit emphasis, I will briefly write $\mathtt{C}_i$ and $\mathtt{S}_i$ instead of $\mathtt{C}_i(n)$ and $\mathtt{S}_i(n)$. Given the solution of the recurrence relation from Section 4.2, we can easily compute $\mathtt{C}_i(n)$ from $\mathtt{c}_i$ by setting $pc_n := \mathbb{E}_n\, \mathtt{c}_i$.

Of course, we get the total number of comparisons and swaps — $\mathtt{c}$ and $\mathtt{s}$ for the first partition, $\mathtt{C}$ and $\mathtt{S}$ for the whole sort — by adding up the frequencies of all corresponding locations:

$$\mathtt{c} := \sum_i \mathtt{c}_i, \qquad \mathtt{s} := \sum_i \mathtt{s}_i, \qquad \mathtt{C} := \sum_i \mathtt{C}_i, \quad \text{and} \quad \mathtt{S} := \sum_i \mathtt{S}_i.$$

4.3.2 On Positions and Values — Some More Notation

The sorting algorithms get as input an array A with entries $A[i]$ for $i = 0, \dots, n$. $A[0]$ contains an element less or equal to any element in the list; we write $A[0] = -\infty$. $A[1], \dots, A[n]$ contain a uniformly chosen random permutation of $[n]$, i. e. more formally, if $\sigma : [n] \to [n]$ is the random permutation, we initially set $A[i] = \sigma(i)$. I will identify a permutation σ with the array A it induces.

Recall that all considered Quicksort variants work in-place. This means, the entries of A are changed during the process of sorting. In the analysis, we will need to refer to the *initial* contents of A, or equivalently to the underlying permutation σ. The **initial array** is denoted by $\mathbf{A_0}$, such that at any time $A_0[i] = \sigma(i)$, whereas potentially $A[i] \neq \sigma(i) = A_0[i]$ if that entry has been changed by the sorting procedure.

The numbers $1, \dots, n$ occur both as **indices / positions** for array accesses and as **elements / values** of the list to be sorted, i. e. as values stored in $A[1], \dots, A[n]$. From the context, it should always be clear whether a given number is an index or an element. Yet, I feel obliged to explicitly warn the reader of possible confusion. In cases where an explicit discrimination of the indices and values is beneficial, I will use the default upright font — e. g. $1, 2$ and S — to denote [sets of] element values, whereas positions and sets thereof will appear in a slanted respectively curly font: $\mathit{1}, \mathit{2}$ and $\mathcal{S}$. Note that p and q are used *both* as values and as indices in the analysis. Then, of course, the "value p" and the "position p" still refer to the same number.

Let us define some commonly used **sets of values**: Let S be the set of all elements smaller than both pivots, M those in the middle and L the large ones, i. e.

$$\begin{aligned} S &:= \{1, \dots, p-1\}, \\ M &:= \{p+1, \dots, q-1\}, \\ L &:= \{q+1, \dots, n\}\,. \end{aligned}$$

Then by Property 1 on page 56, we cannot distinguish $x \in C$ from $y \in C$ for any $C \in \{S, M, L\}$ during the current partitioning step. Hence, for analyzing partitioning costs, we can treat non-pivot elements as **symbolic values** s, m or l when they are elements of S, M or L, respectively. Stated differently: S, M and L are equivalence classes w. r. t. the behavior in the current partitioning step. Obviously, all possible results of the partitioning step correspond to the same word

$$s \cdots s\, p\, m \cdots m\, q\, l \cdots l\,.$$

(This is the definition of the partitioning process!)
Example 4.1 demonstrates the definitions and shows a possible partitioning result of an example permutation.

Example 4.1: Example permutation before … … and after partitioning.

$$\begin{array}{rccccccccc} & p & & & & & & & & q \\ & \boxed{2} & 4 & 7 & 8 & 1 & 6 & 9 & 3 & \boxed{5} \\ \text{value} & p & m & l & l & s & l & l & m & q \end{array} \qquad \begin{array}{ccccccccc} 1 & \boxed{2} & 4 & 3 & \boxed{5} & 6 & 9 & 8 & 7 \\ s & p & m & m & q & l & l & l & l \end{array}$$

Next, we define the **position sets** $\mathcal{S}$, $\mathcal{M}$ and $\mathcal{L}$ as follows:

$$\begin{aligned}\mathcal{S} &:= \{2, \ldots, p\},\\ \mathcal{M} &:= \{p+1, \ldots, q-1\},\\ \mathcal{L} &:= \{q, \ldots, n-1\}\,.\end{aligned}$$

These position sets define three ranges among the non-pivot positions $[2..n-1]$, such that each range contains exactly those positions which are occupied by the corresponding values after partitioning, but before the pivots are swapped to their final place. The right list in Example 4.2 demonstrates this: The set of values at positions $\mathcal{S}$ is exactly S, likewise for $\mathcal{M}$ and $\mathcal{L}$.

Example 4.2: Example permutation before … … after partitioning, but before pivots are swapped in place.

		$\mathcal{S}$	$\mathcal{M}$	$\mathcal{M}$	$\mathcal{L}$	$\mathcal{L}$	$\mathcal{L}$	$\mathcal{L}$	
	$\boxed{2}$	4	7	8	1	6	9	3	$\boxed{5}$
position	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	p	m	l	l	s	l	l	m	q

	$\mathcal{S}$	$\mathcal{M}$	$\mathcal{M}$	$\mathcal{L}$	$\mathcal{L}$	$\mathcal{L}$	$\mathcal{L}$	
$\boxed{2}$	1	4	3	6	9	8	7	$\boxed{5}$
1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
p	s	m	m	l	l	l	l	q

Now, we can formulate the main quantities occurring in the analysis below: For a given permutation, a value type $\mathrm{c} \in \{\mathrm{s}, \mathrm{m}, \mathrm{l}\}$ and a set of positions $\mathcal{P} \subset \{1, \ldots, n\}$, I write $\mathbf{c} \mathbin{@} \mathcal{P}$ for the **number of c-type elements occurring *at positions* in $\mathcal{P}$** of the permutation. Note that we are referring to the permutation or, equivalently, to the *initial* array A_0, not to the 'current' array A. The formal definition is

$$\mathrm{s} \mathbin{@} \mathcal{P} := \Bigl|\bigl\{i \in \mathrm{P} : \mathrm{A}_0[i] \in \mathrm{S}\bigr\}\Bigr|\,.$$

(The definitions for $\mathrm{m} \mathbin{@} \mathcal{P}$ and $\mathrm{l} \mathbin{@} \mathcal{P}$ are similar.)
In Example 4.2, $\mathcal{M} = \{3, 4\}$ holds. At these positions, we find elements 7 and 8 (before partitioning), both belonging to L. Thus, $\mathrm{l} \mathbin{@} \mathcal{M} = 2$, whereas $\mathrm{s} \mathbin{@} \mathcal{M} = \mathrm{m} \mathbin{@} \mathcal{M} = 0$. Likewise, we have $\mathrm{s} \mathbin{@} \mathcal{L} = \mathrm{m} \mathbin{@} \mathcal{L} = 1$ and $\mathrm{l} \mathbin{@} \mathcal{L} = 2$.

Now consider a *random* permutation. Then $\mathrm{c} \mathbin{@} \mathcal{P}$ becomes a random variable. In the analysis, we will encounter the conditional expectation of $\mathrm{c} \mathbin{@} \mathcal{P}$ *given* that the random permutation induces the pivots p and q, i. e. given that the first and last element of the permutation are p and q *or* q and p, respectively. I abbreviate this quantity as $\mathbb{E}_n\,[\mathbf{c} \mathbin{@} \mathcal{P} \,|\, \mathbf{p}, \mathbf{q}]$.

Denote the number of c-type elements by #c. As #c only depends on the value of the pivots, not on the permutation itself, #c is a fully determined constant in $\mathbb{E}_n\,[\mathrm{c} \mathbin{@} \mathcal{P} \,|\, \mathrm{p}, \mathrm{q}]$. In fact, we can directly state

$$\begin{aligned}\#\mathrm{s} &= \;|\mathrm{S}|\; = \mathrm{p} - 1\,,\\ \#\mathrm{m} &= |\mathrm{M}| = \mathrm{q} - \mathrm{p} - 1\,,\\ \#\mathrm{l} &= \;|\mathrm{L}|\; = \mathrm{n} - \mathrm{q}\,.\end{aligned}$$

Hence, given pivots p and q, $c @ \mathcal{P}$ is a *hypergeometrically* distributed random variable: For the c-type elements, we draw their $\#c$ positions out of $n-2$ possible positions via sampling without replacement. Drawing a position in $\mathcal{P}$ is a 'success', a position not in $\mathcal{P}$ is a 'failure'. Accordingly, $\mathbb{E}_n\,[c @ \mathcal{P} \,|\, p, q]$ can be expressed as the mean of this hypergeometric distribution:

$$\mathbb{E}_n\,[c @ \mathcal{P} \,|\, p, q] = \#c \cdot \frac{|\mathcal{P}|}{n-2} \,. \tag{4.13}$$

By the law of total expectation, we finally have

$$\begin{aligned} \mathbb{E}_n\,[c @ \mathcal{P}] &= \sum_{1 \leqslant p < q \leqslant n} \mathbb{E}_n\,[c @ \mathcal{P} \,|\, p, q] \cdot \Pr[\text{pivots } (p, q)] \\ &= \frac{2}{n(n-1)} \sum_{1 \leqslant p < q \leqslant n} \#c \cdot \frac{|\mathcal{P}|}{n-2} \,. \end{aligned}$$

4.3.3 Yaroslavskiy's Partitioning Method

For convenience, the partitioning part of Algorithm 8 is reproduced here. The comparison and swap locations are annotated with the corresponding frequency counters.

```
                $p := A[left];$    $q := A[right]$
$c_0, [s_0]$    if $p > q$ then Swap $p$ and $q$ end if
                $\ell := left + 1;$    $g := right - 1;$    $k := \ell$
                while $k \leqslant g$
$c_1$               if $A[k] < p$
$s_1$                   Swap $A[k]$ and $A[\ell]$
                        $\ell := \ell + 1$
                    else
$c_2$                   if $A[k] \geqslant q$
$c_3$                       while $A[g] > q$ and $k < g$ do $g := g - 1$ end while
$s_2$                       Swap $A[k]$ and $A[g]$
                            $g := g - 1$
$c_4$                       if $A[k] < p$
$s_3$                           Swap $A[k]$ and $A[\ell]$
                                $\ell := \ell + 1$
                            end if
                        end if
                    end if
                    $k := k + 1$
                end while
                $\ell := \ell - 1;$    $g := g + 1$
$s_4$           Swap $A[left]$ and $A[\ell]$
$s_5$           Swap $A[right]$ and $A[g]$
```

4.3.3.1 States After the Outer Loop

As all implementations studied in this chapter, Algorithm 8 uses HOARE's "crossing pointers technique". This technique gives rise to two different cases for "crossing": As the pointers are moved alternatingly towards each other, one of them will reach the crossing point *first* — waiting for the other to arrive.

The asymmetric nature of Algorithm 8 leads to small differences in the number of swaps and comparisons in these two cases: If the left pointer k moves last, we always leave the outer loop of Algorithm 8 with $k = g + 1$ since the loop continues as long as $k \leqslant g$ and k increases by one in each iteration. If g moves last, we decrement g *and* increment k, so we can end up with $k = g + 2$. Consequently, operations that are executed for every value of k experience one additional occurrence. To precisely analyze the impact of this behavior, the following equivalence is useful.

Lemma 4.3: Let $A[1], \ldots, A[n]$ contain a random permutation of $\{1, \ldots, n\}$ for $n \geqslant 2$. Then, Algorithm 8 leaves the outer loop with $k = q + \delta = g + 1 + \delta$ for $\delta \in \{0, 1\}$. (Precisely speaking, the equation holds for the valuations of k, g and q after line 21). Moreover, $\delta = 1$ *iff* initially $A_0[q] > q$ holds, where $q = \max\{A_0[1], A_0[n]\}$ is the large pivot.

In order to proof Lemma 4.3, we need another helper lemma concerning the relation between the array entries in comparisons and the initial entries:

Lemma 4.4: The elements used in the comparisons in lines 6, 10 and 11 have not been changed up to this point. More formally, in lines 6 and 10 holds $A[k] = A_0[k]$ and in line 11, we have $A[g] = A_0[g]$.

Proof. First note that in each iteration of the outer loop, k is the index of a 'fresh' element, since all swaps occur with indices less than k or greater than g. Thus, in line 6 always holds $A[k] = A_0[k]$. As line 10 is the first statement in the else-branch of line 6, this has not changed since the beginning of the iteration. Similarly, in every iteration of the inner loop at line line 11, g refers to a 'fresh' element. So, $A[g] = A_0[g]$ holds there, completing the proof. □

Now, we can tackle the proof of Lemma 4.3.

Proof of Lemma 4.3. The first part is already proven by the discussion above the lemma: We move k and g towards each by at most one entry between two checks, so we always have $k \leqslant g + 2$. We exit the loop once $k > g$ holds. In the end, q is moved to position g in line 24. Just above this line, g has been incremented, so when the loop is left, after line 21, we have $g = q - 1$.

For the 'moreover' part, we show both implications separately. Assume first that $\delta = 1$, i. e. the loop is left with a difference of $\delta + 1 = 2$ between k and g. This difference can only show up when both k is incremented *and* g is decremented in the last iteration. Hence, in this last iteration we must have entered the else-if-branch in line 10 and accordingly $A[k] \geqslant q$ must have held there — and by Lemma 4.4 also $A_0[k] \geqslant q$. I claim that in fact even the strict inequality $A_0[k] > q$ holds. To see this, note that if $k < n$, we have

$A_0[k] \neq A_0[n] = q$ as we assume distinct elements. This already implies $A_0[k] > q$. Now assume towards a contradiction, $k = n$ holds in the last execution of line 10. Since g is initialized in line 4 to $\mathit{right} - 1 = n - 1$ and is only decremented in the loop, we have $g \leqslant n - 1$. But this is a contradiction to the loop condition "$k \leqslant g$": $n = k \leqslant g \leqslant n - 1$. So, we have shown that $A_0[k] > q$ for the last execution of line 10.

By assumption, $\delta = 1$, so $k = q + 1$ upon termination of the loop. As k has been incremented once since the last test in line 10, we find $A_0[q] > q$ there, as claimed.

Now, assume conversely that $A_0[q] > q$ holds. As g stops at $q - 1$ and is decremented in line 13, we have $g = q$ for the last execution of line 11. Using the assumption and Lemma 4.4 yields $A[g] = A[q] = A_0[q] > q$. Thus, the loop in line 11 must have been left because of a violation of "$k < g$", the second part of its loop condition. The violation implies $k \geqslant g = q$ in line 12. With the following decrement of g and increment of k, we leave the loop with $k \geqslant g + 2$, so $\delta = 1$. □

Lemma 4.3 allows to compute the probability for the event $\delta = 1$:

Corollary 4.5: $\delta = 1$ occurs with conditional probability $\frac{n-q}{n-2}$ given that the large pivot is q, for $n \geqslant 3$. Consequently, $\mathbb{E}_n\,[\delta \mid p, q] = \frac{n-q}{n-2}$.

Proof. Using Lemma 4.3, we only need to consider $A_0[q]$. We do a case distinction.

For $q < n$, $A_0[q]$ is one of the non-pivot elements. (We have $1 \leqslant p < q < n$.) Any of the $n - 2$ non-pivot elements can take position $A_0[q]$, and among those, $n - q$ elements are $> q$. This gives a probability of $\frac{n-q}{n-2}$ for $A_0[q] > q$.

For $q = n$, q is the maximum of all elements in the list, so we cannot possibly have $A_0[q] > q$. This implies a probability of $0 = \frac{n-q}{n-2}$. □

Corollary 4.6: $\mathbb{E}_n\,\delta = \frac{1}{3}$

Proof. The proof is by computing:

$$\begin{aligned}
\mathbb{E}\,\delta &= \sum_{1 \leqslant p < q \leqslant n} \Pr[\text{pivots}\,(p,q)] \cdot \mathbb{E}\,[\delta \mid p, q] \\
&= \sum_{1 \leqslant p < q \leqslant n} \tfrac{2}{n(n-1)} \cdot \tfrac{n-q}{n-2} \\
&= \tfrac{2}{n(n-1)(n-2)} \Bigg(\sum_{1 \leqslant p < q \leqslant n} n - \sum_{1 \leqslant p < q \leqslant n} q \Bigg) \\
&\underset{(\Sigma c),(\mathbb{E}q)}{=} \tfrac{2}{n(n-1)(n-2)} n\tbinom{n}{2} - \frac{\frac{2}{3}(n+1)}{n-2} \\
&= \frac{n}{n-2} - \frac{2(n+1)}{3(n-2)} = \frac{3n - 2n - 2}{3(n-2)} \\
&= \tfrac{1}{3}\,.
\end{aligned}$$

□

4.3.3.2 c_0 in Algorithm 8

Line 3 — which corresponds to c_0 — is executed exactly once in the partitioning step, no matter how the list looks like. Hence

$$c_0 = 1 \ . \tag{4.14}$$

Note however, that we skip the whole partitioning step if *right* − *left* ∈ $\{-1, 0\}$, as corresponding (sub) lists have length $\leqslant 1$ and are thus already sorted by definition.

4.3.3.3 c_1 in Algorithm 8

Line 6, corresponding to c_1, is the first statement in the outer loop of Algorithm 8. So, we execute this comparison for *every value* variable k attains — except for the last value of k, since we increment k at the end of the loop body in line 20 and then leave the loop.

k is initialized to *left* + 1 = 2 in line 4 and by Lemma 4.3, it stops with $k = q + \delta$ with $\delta \in \{0, 1\}$. This means, at line 6, k attains all values in

$$\mathcal{K} := \{2, \ldots, q + \delta - 1\} \ . \tag{4.15}$$

For c_1, this means $c_1 = |\mathcal{K}| = q - 2 + \delta$ and by Corollary 4.5 we find

$$\begin{aligned} \mathbb{E}_n\,[c_1 \mid p, q] &= q - 2 + \mathbb{E}_n\,[\delta \mid p, q] \\ &= q - 2 + \tfrac{n-q}{n-2} \ . \end{aligned}$$

By the law of total expectation, we can compute the unconditional expected value

$$\begin{aligned} \mathbb{E}_n\, c_1 &= \sum_{1 \leqslant p < q \leqslant n} \Pr[\text{pivots}\,(p, q)] \cdot \mathbb{E}_n\,[c_1 \mid p, q] \\ &= \tfrac{2}{n(n-1)} \sum_{1 \leqslant p < q \leqslant n} \left(q - 2 + \tfrac{n-q}{n-2}\right) \\ &= \tfrac{2}{n(n-1)} \sum_{1 \leqslant p < q \leqslant n} q \quad - \quad \tfrac{2}{n(n-1)} \sum_{1 \leqslant p < q \leqslant n} 2 \ + \ \mathbb{E}_n\,\delta \\ &\underset{\text{Cor. 4.6,}(\mathbb{E}q),(\mathbb{E}c)}{=} \tfrac{2}{3}(n+1) - 2 \ + \ \tfrac{1}{3} \\ &= \tfrac{2}{3}n - 1 \end{aligned} \tag{4.16}$$

4.3.3.4 c_2 in Algorithm 8

The last two counters were rather straight-forward to compute. This time it will get a little harder. c_2 corresponds to line 10, which is the first line in the else-branch. Consequently, line 10 is reached each time the check in line 6 fails. This immediately tells us $c_2 \leqslant c_1$.

In Section 4.3.3.3, I argued that $c_1 = |\mathcal{K}|$ for $\mathcal{K} = \{2, \ldots, q + \delta - 1\}$. The condition of line 6 is "$A[k] < p$", so we reach line 10 for every value $k \in \mathcal{K}$ with $A[k] \geqslant p$. As we assume all elements to be distinct and p and q lie outside the range $\mathcal{K}$,[16] the case $\mathrm{A}[k] \in \{\mathrm{p}, \mathrm{q}\}$ cannot happen, so $\mathrm{c}_2 = \bigl|\{k \in \mathcal{K} : \mathrm{A}[k] \in \mathrm{S} \cup \mathrm{M}\}\bigr|$. Applying Lemma 4.4 we can replace A with A_0. Hence, with the definitions from Section 4.3.2, we know

$$\mathrm{c}_2 = \mathrm{m}@\mathcal{K} \;+\; \mathrm{l}@\mathcal{K}\,.$$

Note that we have two random variables here: $\mathcal{K}$ is random because it depends on δ and $\mathrm{c}@\mathcal{K}$ for $\mathrm{c} \in \{\mathrm{m}, \mathrm{l}\}$ is random as it depends on the permutation *and* on $\mathcal{K}$. So, the random variables are *not* stochastically *independent* and we cannot simply replace them by their expectation. Luckily, Lemma 4.3 allows us to resolve the dependence by a case distinction:

If $\delta = 1$, $\mathcal{K}$ includes as last value the index $\mathrm{q} + \delta - 1 = \mathrm{q}$. By Lemma 4.3, $\mathrm{A}_0[\mathrm{q}] > \mathrm{q}$. This means, we get *for sure* a contribution to $\mathrm{l}@\mathcal{K}$ and *for sure no* contribution to $\mathrm{m}@\mathcal{K}$. If we define $\mathcal{K}' := \{2, \ldots, q-1\}$, we make this more explicit: $\mathrm{l}@\mathcal{K} = \mathrm{l}@\mathcal{K}' + 1$ and $\mathrm{m}@\mathcal{K} = \mathrm{m}@\mathcal{K}'$.

If $\delta = 0$, we have $\mathcal{K} = \mathcal{K}'$. Hence, in this case, we trivially have $\mathrm{c}@\mathcal{K} = \mathrm{c}@\mathcal{K}'$ for any $\mathrm{c} \in \{\mathrm{s}, \mathrm{m}, \mathrm{l}\}$. Putting both cases together, we find

$$\mathrm{c}_2 = \mathrm{m}@\mathcal{K}' + \mathrm{l}@\mathcal{K}' + \delta\,.$$

For the conditional expected value with given pivots, we know by eq. (4.13) on page 70:

$$\begin{aligned}\mathbb{E}\,[\mathrm{c}_2 \mid \mathrm{p}, \mathrm{q}] &= \mathbb{E}\left[\mathrm{m}@\mathcal{K}' \,|\, \mathrm{p}, \mathrm{q}\right] + \mathbb{E}\left[\mathrm{l}@\mathcal{K}' \,|\, \mathrm{p}, \mathrm{q}\right] + \mathbb{E}\,[\delta \mid \mathrm{p}, \mathrm{q}] \\ &= \bigl((\mathrm{q}-\mathrm{p}-1) + (\mathrm{n}-\mathrm{q})\bigr)\frac{\mathrm{q}-2}{\mathrm{n}-2} \;+\; \mathbb{E}\,[\delta \mid \mathrm{p}, \mathrm{q}]\end{aligned}$$

Computing the total expectation is again a means of elementary summations

$$\begin{aligned}\mathbb{E}\,\mathrm{c}_2 &= \mathbb{E}\left[\mathrm{m}@\mathcal{K}'\right] + \mathbb{E}\left[\mathrm{l}@\mathcal{K}'\right] + \mathbb{E}\,\delta \\ &\underset{\text{Cor. 4.6}}{=} \tfrac{2}{\mathrm{n}(\mathrm{n}-1)} \sum_{1 \leqslant \mathrm{p} < \mathrm{q} \leqslant \mathrm{n}} (\mathrm{n}-\mathrm{p}-1)\frac{\mathrm{q}-2}{\mathrm{n}-2} + \tfrac{1}{3} \\ &= \left(\tfrac{2}{\mathrm{n}(\mathrm{n}-1)} \sum_{1 \leqslant \mathrm{p} < \mathrm{q} \leqslant \mathrm{n}} \bigl((\mathrm{n}-1)\mathrm{q} - \mathrm{pq} + 2\mathrm{p} - 2(\mathrm{n}-1)\bigr)\right)\Big/(\mathrm{n}-2) + \tfrac{1}{3} \\ &\underset{(\mathbb{E}\mathrm{q}),(\mathbb{E}\mathrm{p}),(\mathbb{E}\mathrm{p}\,\mathrm{q}),(\mathbb{E}\mathrm{c})}{=} \frac{(\mathrm{n}-1)\frac{2}{3}(\mathrm{n}+1) - \frac{1}{12}(\mathrm{n}+1)(3\mathrm{n}+2) + \frac{2}{3}(\mathrm{n}+1) - 2(\mathrm{n}-1)}{\mathrm{n}-2} + \tfrac{1}{3} \\ &= \frac{\frac{1}{12}(\mathrm{n}-2)(5\mathrm{n}-11)}{\mathrm{n}-2} + \tfrac{1}{3} \\ &= \tfrac{5}{12}\mathrm{n} - \tfrac{7}{12}\,. \end{aligned} \tag{4.17}$$

[16] Initially, p and q are located at $\mathrm{A}_0[1]$ and $\mathrm{A}_0[n]$, respectively. $1 \notin \mathcal{K}$ by definition. The precise argument why $n \notin \mathcal{K}$ is as follows: If n were in $\mathcal{K}$, then $n \leqslant \mathrm{q} + \delta - 1$. This is only possible for, $\mathrm{q} = \mathrm{n}$ and $\delta = 1$. But by Lemma 4.3, $\delta = 1$ implies $\mathrm{A}_0[q] > \mathrm{q} = \mathrm{n}$, which is a contradiction to $\mathrm{A}_0[q] \in [\mathrm{n}]$.

4.3.3.5 c_3 in Algorithm 8

Although line 11 belonging to c_3 seems to be buried deep in the structure of Algorithm 8, the analysis is not that nasty. First note, that inside the outer loop, the value of g is only changed in the inner loop body of line 11 and in line 13. This means that we *always* decrement g *after* comparison c_3 and only then. Hence, we execute the comparison in line 11 for all values g attains in the loop except for the last one.

In line 4, g is initialized to $\mathit{right} - 1 = n - 1$ and by Lemma 4.3, we leave the loop with $g = q - 1$. So, in line 11 g takes the values

$$\mathcal{G} := \{n-1, n-2, \dots, q\} .$$

Accordingly, we know $c_3 = |\mathcal{G}| = n - q$. For given q, c_3 is actually constant, so trivially $\mathbb{E}\,[c_3 \mid p, q] = n - q$. Finally, we can compute the total expectation using Eqs. ($\mathbb{E}$c) and ($\mathbb{E}$q):

$$\begin{aligned} \mathbb{E}\,c_3 &= \tfrac{2}{n(n-1)} \sum_{1 \leqslant p < q \leqslant n} (n-q) \\ &= n - \tfrac{2}{3}(n+1) \\ &= \tfrac{1}{3}n - \tfrac{2}{3} \,. \end{aligned} \tag{4.18}$$

4.3.3.6 c_4 in Algorithm 8

Frequency c_4 corresponds to line 14 of Algorithm 8, which belongs to the else-if-branch starting in line 10. Consequently, line 14 is reached exactly once for every comparison at c_2 with result *true*. By the way, the same holds true for the swap frequency s_2 in line 12.

I already showed in Section 4.3.3.4 that $c_2 = m@\mathcal{K}' + l@\mathcal{K}' + \delta$. Recall that $\mathcal{K}' = \{2, \dots, q-1\}$. Of these executions of the comparison "$A[k] \geqslant q$" in line 10, all those with a key from L yield *true*.[17] Among the indices from $\mathcal{K}'$, this happens for $l@\mathcal{K}'$ positions by definition. The additional contribution to c_2 resulted from the Case $\delta = 1$ (cf. Section 4.3.3.1), where we got the additional index $k = q$. By Lemma 4.3 in Case $\delta = 1$, we have $A_0[q] > q$, so we enter the else-if-branch *for sure* and c_4 inherits the contribution of δ from c_2. Putting everything together, we proved

$$c_4 = l@\mathcal{K}' + \delta \,.$$

Using eq. (4.13) and Corollary 4.6, we find

$$\begin{aligned} \mathbb{E}\,[c_4 \mid p, q] &= \mathbb{E}\left[l@\mathcal{K}' \,|\, p, q\right] + \mathbb{E}\,[\delta \mid p, q] \\ &= (n-q)\frac{q-2}{n-2} + \frac{n-q}{n-2} \,. \end{aligned}$$

The law of total probability finally gives

$$\begin{aligned} \mathbb{E}\,c_4 &= \mathbb{E}\left[l@\mathcal{K}'\right] + \mathbb{E}\,\delta \\ &= \tfrac{1}{6}(n-3) + \tfrac{1}{3} \\ &= \tfrac{1}{6}n - \tfrac{1}{6} \,. \end{aligned} \tag{4.19}$$

[17] As discussed in Section 4.3.3.4, $n \notin \mathcal{K}'$, so $A[k] \geqslant q$ is equivalent to $A[k] > q$, here.

4.3.3.7 s_0 in Algorithm 8

Frequency s_0 corresponds to the swap of the two pivots in line 3. It is guarded by the comparison "$p > q$" in the same line, i. e. we execute the swap as often as c_0, where the result is *true*: $s_0 = c_0 \cdot [p > q]$. In the random permutation model, $\Pr[p > q] = \frac{1}{2}$ for two distinct elements, so by $c_0 = 1$ from eq. (4.14) we have

$$\mathbb{E}\, s_0 = \tfrac{1}{2} \,. \tag{4.20}$$

4.3.3.8 s_1 in Algorithm 8

Frequency s_1 comprises the swap in line 7, which is contained in the if-branch following the comparison "$A[k] < p$" in line 6. We recall from Section 4.3.3.3 that line 6 is executed for every $k \in \mathcal{K}$ with $\mathcal{K} = \{2, \ldots, q + \delta - 1\}$, so $s_1 = s @ \mathcal{K}$.

Similar to the situation for c_2, we have a hidden stochastic dependency, here: Both $s @ \mathcal{K}$ and $\mathcal{K}$ itself are random, since $\mathcal{K}$ depends on δ. The dependency is resolved by a case distinction following Lemma 4.3. For $\delta = 1$, $\mathcal{K} = \mathcal{K}' \,\dot\cup\, \{q\}$ with $\mathcal{K}' = \{2, \ldots, q - 1\}$ as in Section 4.3.3.4. Now by Lemma 4.3, $A_0[q] > q$ and we do *not* get a contribution to $s @ \mathcal{K}$ for position q. This means $s @ \mathcal{K} = s @ \mathcal{K}'$ for $\delta = 1$. For the case $\delta = 0$, we have $\mathcal{K} = \mathcal{K}'$ altogether, so we conclude

$$s_1 = s @ \mathcal{K}' \,. \tag{4.21}$$

Expected values are then computed as usual

$$\begin{aligned}
\mathbb{E}\,[s_1 \mid p, q] &= \mathbb{E}\left[s @ \mathcal{K}' \,|\, p, q\right] \\
&\underset{(4.13)}{=} (p-1)\frac{q-2}{n-2} \,. \\
\mathbb{E}\, s_1 &= \tfrac{2}{n(n-1)} \sum_{1 \leqslant p < q \leqslant n} (p-1)\frac{q-2}{n-2} \\
&= \tfrac{1}{4}n - \tfrac{5}{12} \,.
\end{aligned} \tag{4.22}$$

4.3.3.9 s_2 in Algorithm 8

After every execution of line 12 corresponding to s_2, we reach line line 14 exactly once. The frequency of the latter has already been computed as c_4, see Section 4.3.3.6. We rejoice in the saved work and conclude with $s_2 = c_4$ and copy

$$\begin{aligned}
\mathbb{E}\, s_2 &= \mathbb{E}\, c_4 \\
&\underset{(4.19)}{=} \tfrac{1}{6}n - \tfrac{1}{6} \,.
\end{aligned} \tag{4.23}$$

4.3.3.10 s_3 in Algorithm 8

The counter s_3 corresponds to line 15. The following observation is the key to determine s_3: Variable ℓ is only changed inside the loop at lines 8 and 16, where it is incremented by one. Both of these lines immediately follow a swap, namely in lines 7 and 15. Consequently, if ℓ

attains the values $\mathcal{L}$ in the loop, we get $s_1 + s_3 = |\mathcal{L}| - 1$ swaps for lines 7 and 15. Since we know s_1 from Section 4.3.3.8, we can use this to determine s_3 from $|\mathcal{L}|$.

It remains to determine $\mathcal{L}$. In line 4, ℓ is initialized to $\mathit{left} + 1 = 2$. Moreover, line 23 places the small pivot p at position ℓ, so by the correctness of Algorithm 8, we must have $\ell = p$ at line 23. Just above, ℓ is decremented, so we leave the outer loop at line 21 with $\ell = p + 1$. We find $\mathcal{L} = \{2, \ldots, p + 1\}$ and by eq. (4.21) for s_1:

$$s_3 = |\mathcal{L}| - 1 - s_1 = p - 1 \; - \; s@\mathcal{K}' \;.$$

(Recall $\mathcal{K}' = \{2, \ldots, q - 1\}$.)
By linearity of $\mathbb{E}$, this relation translates to expected values:

$$\begin{aligned} \mathbb{E}\, s_3 &= \mathbb{E}\, p - 1 - \mathbb{E}\, s_1 \\ &\underset{(\mathbb{E}p),(4.22)}{=} \tfrac{1}{3}(n+1) - 1 - \left(\tfrac{1}{4}n - \tfrac{5}{12}\right) \\ &= \tfrac{1}{12}n - \tfrac{1}{4} \;. \end{aligned} \tag{4.24}$$

4.3.3.11 s_4 and s_5 in Algorithm 8

The last two swaps in lines 23 and 24 are both executed exactly once per partitioning step. This is the same as for c_0, see Section 4.3.3.2. We have

$$s_4 = s_5 = 1 \;. \tag{4.25}$$

4.3.4 Sedgewick's Partitioning Method

As in the last section, the partitioning part of Algorithm 7 is reproduced here for convenience. The comparison and swap locations are annotated with the corresponding frequency counters.

```
                 $i := left$;      $i_1 := left$
                 $j := right$;     $j_1 := right$
                 $p := A[left]$;   $q := A[right]$
$c_0, [s_0]$     if $p > q$ then Swap $p$ and $q$ end if
                 while true
                     $i := i + 1$
$c_1$                while $A[i] \leqslant q$
                         if $i \geqslant j$ then break outer while end if
$c_2$                    if $A[i] < p$
$s_1$                        $A[i_1] := A[i]$; $i_1 := i_1 + 1$; $A[i] := A[i_1]$
                         end if
                         $i := i + 1$
                     end while
                     $j := j - 1$
$c_3$                while $A[j] \geqslant p$
$c_4$                    if $A[j] > q$
$s_2$                        $A[j_1] := A[j]$; $j_1 := j_1 - 1$; $A[j] := A[j_1]$
                         end if
                         if $i \geqslant j$ then break outer while end if
                         $j := j - 1$
                     end while
$s_3$                $A[i_1] := A[j]$; $A[j_1] := A[i]$
|                    $i_1 := i_1 + 1$;  $j_1 := j_1 - 1$
|                    $A[i] := A[i_1]$; $A[j] := A[j_1]$
                 end while
$s_4$            $A[i_1] := p$
$s_5$            $A[j_1] := q$
```

Note that in Algorithm 7, the elements are not directly exchanged, but written back one position apart from the old position of their swap partner. Thereby, we do not need a temporary storage for one of the elements. As the runtime contributions are nearly the same as for a direct exchange, we ignore the difference in our terminology and call such a sequence of element movements a 'swap'. In Chapter 7, I will explicitly count low level instructions, making up for the sloppiness of this chapter.

4.3.4.1 The Crossing Point of i and j

As for Algorithm 8, we need to state some conditions on when array elements are first modified. Therefore, the following lemma relates current entries in A with the initial values of A_0:

Lemma 4.7: Except for the pivots, the elements used in the comparisons in lines 8, 10, 16 and 17 have not been changed up to this point. More formally, in lines 8 and 10 holds $A[i] = A_0[i] \vee A[i] \in \{p, q\}$ and in lines 16 and 17, we have $A[j] = A_0[j] \vee A[j] \in \{p, q\}$.

Proof. After a swap involving $A[i]$ and $A[j]$, we always increment i respectively decrement j. So, both point to an element not yet changed when the comparisons in lines 8 and 16 are done. The pivot elements form an exception to this rule since they might have been swapped before the loop in line 5. □

Algorithm 7 uses the crossing pointers i and j. i starts from the left end of the list and moves right, j vice versa. The checks in lines 9 and 20 ensure that we leave the outer loop as soon as $i = j$. Therefore, we always have $i = j =: \chi$ when we leave the outer loop at line 26. χ will be called the **crossing point** of i and j. For determining the frequencies of comparison and swap locations, it is vital to know χ. Unfortunately, its detailed value is somewhat intricate.

Lemma 4.8: Let array $A[1], \ldots, A[n]$ contain a random permutation of $\{1, \ldots, n\}$ for $n \geqslant 2$. $p = \min\{A_0[1], A_0[n]\}$ and $q = \max\{A_0[1], A_0[n]\}$ are the two chosen pivots.
Then, the crossing point χ of i and j is the index of the $(p+1)$*st non-*m*-type element* in A_0.
More formally, define $\overrightarrow{\neg m}(x)$, the "not-m-index of x", to be the number of not-m-type elements left of x in A_0:

$$\overrightarrow{\neg m}(x) := \Big|\big\{A_0[y] : 1 \leqslant y \leqslant x\big\} \cap \big([1..n] \setminus M\big)\Big| \;.$$

(Recall $M = [p+1..q-1]$ is the value set of m-type elements; see Section 4.3.2).
Then, χ is the smallest position with not-m-index $p+1$:

$$\chi = \min\big\{x : \overrightarrow{\neg m}(x) = p+1\big\} \;.$$

Remark on Lemma 4.8

> For $q = p+1$, there are no m-type elements and the not-m-index coincides with the ordinary index in A. The reader might find it instructive to read the proof for this special case by mentally replacing all occurrences of $\overrightarrow{\neg m}(x)$ by x.

Proof. First, we show that the crossing point cannot be on an m-type element:

$$A_0[\chi] \notin M \;. \tag{4.26}$$

If we leave the loop via line 9, then either $A[j] = q \notin M$ — if j has not moved yet — or $A_0[j] = A[j] < p$ as the last j-loop was left violating the condition in line 16. $A_0[j] = A[j]$

follows from Lemma 4.7 for the second case. In the first case, we have $j = n$, so $A_0[j]$ is either p or q, both of which are not in M. Similarly, if we leave the outer loop from line 20, then the i-loop above was left by $A_0[i] > q$.

The next step is to prove that when we leave the outer loop, we have

$$\overrightarrow{\neg m}(i) = i_1 + 1 \ . \tag{4.27}$$

To this end, consider the inner loop starting at line 8. i attains all values in $\{2, \ldots, \chi\}$ there and for each such $i < \chi$, $A[i] = A_0[i]$ is either an s, m or l element. m-elements always stay between i_1 and i. For $i < \chi$, s-elements are swapped behind i_1 in line 11 and for each l-element, the second inner loop finds an s-partner which is swapped behind i_1 in line 23. Each such swap increments i_1. For the last element $i = \chi$, the corresponding swaps are not reached. By (4.26), $A[\chi] \notin M$. If $A[\chi] \in S$, we enter the i-loop, but then break the outer loop at line 9, before the swap in line 11. Similarly if $A[\chi] \in L$, we skip the i-loop, but break inside the j-loop, such that the swap in line 23 is not reached. Together, this means i_1 is the number of not-m-elements left of χ in A_0 minus the sure one at $A_0[\chi]$ itself, so (4.27) follows.

Finally, i_1 stops with $i_1 = p$ as p is swapped there in line 27. So, at the end

$$\overrightarrow{\neg m}(\chi) = \overrightarrow{\neg m}(i) \underset{(4.27)}{=} i_1 + 1 = p + 1 \ .$$

$A_0[\chi] \notin M$ means that the not-m-index *increases at* χ, so χ is the smallest index with $\overrightarrow{\neg m}(\chi) = p + 1$. □

In addition to the deepened understanding of Algorithm 7, Lemma 4.8 also allows to compute the expected value of χ:

Proposition 4.9: For the crossing point χ of i and j holds

$$\mathbb{E}\,[\chi \mid p, q] = p + 1 \ + \ (q - p - 1)\tfrac{p}{p+n-q} \ ,$$
$$\mathbb{E}\,\chi = \tfrac{1}{2}n + \tfrac{3}{2} - \tfrac{1}{n} \ .$$

Proof. By Lemma 4.8, we know $\chi = p + 1 + \mu$ for μ the number of m-type elements left of the $(p+1)$st non-m-element. Assume now, A_0 contains a random permutation with given pivots $p < q$. Then, trivially, $0 \leqslant \mu \leqslant |M| = q - p - 1$. By linearity of the expectation, $\mathbb{E}\,[\chi \mid p, q] = p + 1 + \mathbb{E}\,[\mu \mid p, q]$.

To determine $\mathbb{E}\,[\mu \mid p, q]$, we imagine the following process of creating all permutations with pivots p and q. First, take the elements from S and L and arrange them in random order. There are $\#sl := |S \cup L| = p - 1 + n - q$ such elements and $\#sl + 1$ different places, where insertions are possible — left and right of the elements and between any two. Now, take the elements from M in random order and choose a *multiset* of $|M| = q - p - 1$ insertion slots among the $\#sl + 1$ ones. Finally, successively insert the m-elements into the slots corresponding to the sorted multiset. After each insertion, the slot moves *behind* the newly inserted element.

The process is probably best understood via an example, so let us execute this construction once for $n = 9$, $p = 4$ and $q = 8$. We start with $S \cup L = \{1, 2, 3, 9\}$ in random order: ␣2␣9␣3␣1␣. The $\#sl = 4$ elements induce $\#sl + 1 = 5$ slots for possible insertions, written as ␣. Now we choose some order for $M = \{5, 6, 7\}$, say $5, 7, 6$. Then, we choose a multiset of slots with cardinality $|M| = 3$; suppose we choose the second slot twice and the last one once. Now, all necessary decisions for the insertions have been taken:

␣2␣9␣3␣1␣ → ␣25␣9␣3␣1␣ → ␣257␣9␣3␣1␣ → ␣257␣9␣3␣16␣

Note that throughout the process, we have $\#sl + 1$ slots and that the relative order among the m-elements is preserved. Finally, attach p and q at beginning and end — again in random order. For example ␣257␣9␣3␣16␣ becomes 825793164.

If all random choices are done uniformly, the resulting permutation is uniformly distributed among all with pivots p and q: We create each such permutation in exactly one way as the shuffle of elements from $S \cup L$ and elements from M, retaining relative order.

Recall that μ is the number of m-type elements left of the $(p + 1)$st non-m-element. The above process reveals the behavior of μ: There are exactly p slots left of the $(p + 1)$st non-m-element. For the example from above, $\mathfrak{i}$ and $\mathfrak{j}$ will cross on the 1 and we have $p = 4$ slots left of the crossing point: $8\overset{\times}{\text{␣}}2\overset{\times}{\text{␣}}9\overset{\times}{\text{␣}}3\overset{\times}{\text{␣}}1\text{␣}4$.

Any m-element that is inserted into one of these slots contributes to μ. Their number of binomially distributed: We draw $|M|$ balls with replacement from an urn with $\#sl + 1$ balls in total, p of which are red. μ is the total number of red balls drawn. Consequently, $\mathbb{E}\,[\mu \mid p, q]$ is the mean of this binomial distribution:

$$\mathbb{E}\,[\mu \mid p, q] = |M| \frac{p}{\#sl + 1} = (q - p - 1) \frac{p}{p + n - q} \; .$$

By the law of total expectation and the linearity of $\mathbb{E}$, we can express the unconditional expectation of χ as

$$\begin{aligned} \mathbb{E}\,\chi &= \mathbb{E}\,p + \mathbb{E}\,1 + \mathbb{E}\,\mu \\ &\underset{(\mathbb{E}c),(\mathbb{E}p)}{=} \tfrac{1}{3}(n+1) + 1 + \mathbb{E}\,\mu \; . \end{aligned}$$

$\mathbb{E}\,\mu$ can be computed from $\mathbb{E}\,[\mu \mid p, q]$:

$$\begin{aligned} \mathbb{E}\,\mu &= \sum_{1 \leqslant p < q \leqslant n} \Pr[\text{pivots}\, p, q] \cdot \mathbb{E}\,[\mu \mid p, q] \\ &= \tfrac{2}{n(n-1)} \sum_{1 \leqslant p < q \leqslant n} (q - p - 1) \tfrac{p}{p+n-q} \\ &= \tfrac{2}{n(n-1)} \sum_{p=1}^{n-1} p \sum_{m=0}^{n-p-1} \tfrac{m}{(n-1)-m} \\ &= \tfrac{2}{n(n-1)} \sum_{p=1}^{n-1} p \sum_{m=0}^{n-p-1} \left(-1 + \tfrac{n-1}{(n-1)-m}\right) \end{aligned}$$

$$\begin{aligned}
&= -\tfrac{2}{n(n-1)} \sum_{p=1}^{n-1} p(n-p) + \tfrac{2}{n} \sum_{p=1}^{n-1} p \sum_{m=p}^{n-1} \tfrac{1}{m} \\
&= -\tfrac{2}{n(n-1)} \sum_{p=1}^{n-1} p(n-p) + \tfrac{2}{n} \sum_{p=1}^{n-1} p(\mathcal{H}_{n-1} - \mathcal{H}_{p-1}) \\
&\underset{(\mathbb{E}p)}{=} -\tfrac{1}{3}(n+1) + \mathcal{H}_{n-1}(n-1) - \tfrac{2}{n} \sum_{p=1}^{n-1} (p-1)\mathcal{H}_{p-1} - \tfrac{2}{n} \sum_{p=1}^{n-1} \mathcal{H}_{p-1} \\
&\underset{(\Sigma i\mathcal{H}_i)}{=} -\tfrac{1}{3}(n+1) + \mathcal{H}_{n-1}(n-1) - \tfrac{2}{n}\tbinom{n-1}{2}(\mathcal{H}_{n-1} - \tfrac{1}{2}) - \tfrac{2}{n}(n-1)(\mathcal{H}_{n-1} - 1) \\
&= -\tfrac{1}{3}(n+1) + \mathcal{H}_{n-1}\underbrace{\big(n-1-\tfrac{(n-1)(n-2)}{n} - \tfrac{2(n-1)}{n}\big)}_{=0} \\
&\qquad + \tfrac{1}{2n}\big(\underbrace{(n-1)(n-2)+4(n-1)+2}_{=n(n+1)}-2\big) \\
&= \tfrac{1}{6}(n+1) - \tfrac{1}{n}
\end{aligned}$$

These lengthy rearrangements yield $\mathbb{E}\,\mu = \frac{1}{6}(n+1) - \frac{1}{n}$, so that we finally find

$$\begin{aligned}
\mathbb{E}\,\chi &= \tfrac{1}{3}(n+1)+1 \quad + \quad \tfrac{1}{6}(n+1) - \tfrac{1}{n} \\
&= \tfrac{1}{2}n + \tfrac{3}{2} - \tfrac{1}{n} \;.
\end{aligned}$$

□

4.3.4.2 c_0 in Algorithm 7

Line 5 corresponds to c_0 and is executed exactly once per partitioning step. Hence

$$c_0 = 1 \;. \tag{4.28}$$

4.3.4.3 c_1 in Algorithm 7

Frequency c_1 corresponds to line 8. To determine how often this line is reached, we consider the locations where the value of i is changed. In fact, after initialization in line 2, i is only incremented — namely in lines 7 and 13. Now, every execution of line 8 is immediately preceded by one of those two increments. So, line 8 is reached once for every value of i except the initialization value *left* $= 1$. In Section 4.3.4.1, the largest value attained by i was called χ, so with

$$\mathcal{I} := \{2, \ldots, \chi\} \tag{4.29}$$

we can state $c_1 = |\mathcal{I}| = \chi - 1$. The conditional and total expected values follow from Proposition 4.9 and the linearity of the expectation:

$$\begin{aligned}\mathbb{E}\,[c_1 \mid p, q] &= \mathbb{E}\,[\chi \mid p, q] - 1\\ &= p + \frac{(q-p-1)p}{p+n-q}\\ \mathbb{E}\, c_1 &= \mathbb{E}\,\chi - 1\\ &= \tfrac{1}{2}n + \tfrac{1}{2} - \tfrac{1}{n}\,. \end{aligned} \tag{4.30}$$

4.3.4.4 c_2 in Algorithm 7

Comparison marker c_2 corresponds to line 10, which is located inside the first inner loop. Thus, it is only reached if $A[i] \leqslant q$. Moreover, since the check for $i \geqslant j$ in line 9 is above line 10, we do not reach line 10 for the last value of i, namely χ. This ensures that $A[i] = q$ cannot happen. With

$$\mathcal{I}' := \mathcal{I}\backslash\{\chi\} \underset{(4.29)}{=} \{2, \ldots, \chi - 1\}, \tag{4.31}$$

we hence find $c_2 = s\,@\,\mathcal{I}' + m\,@\,\mathcal{I}'$. As for c_2 in Algorithm 8, we have a nasty hidden dependence here: Both $\mathcal{I}'$ and $c\,@\,\mathcal{I}'$ are random variables and they depend non-trivially on each other. Therefore, we can not directly compute the expectation of c_2 from $s\,@\,\mathcal{I}' + m\,@\,\mathcal{I}'$ using eq. (4.13) and Proposition 4.9.

Let us tackle the two summands separately and start with $s\,@\,\mathcal{I}'$. Since we count s-type elements here, no position $x \in \mathcal{I}'$ with $A_0[x] \in M$ contributes. Therefore, we can restrict our view entirely to positions of *non-m-type elements*. There are $|S \cup L| = (p-1) + (n-q)$ such positions in total, excluding the pivot positions. But how many of those are contained in $\mathcal{I}'$? Luckily, Lemma 4.8 provides the answer: χ is the first index with $p+1$ non-m-elements left of it. $\mathcal{I}'$ does not contain positions 1 and χ, both of which contain a non-m-element: $A_0[1] \in \{p, q\}$ and $A_0[\chi] \notin M$ by eq. (4.26) from the proof of Lemma 4.8. Ergo, $p-1$ of the "not-m-positions" are contained in $\mathcal{I}'$ and we conclude by eq. (4.13)

$$\mathbb{E}\left[s\,@\,\mathcal{I}' \,|\, p, q\right] = (p-1)\frac{p-1}{(p-1)+(n-q)}$$

For $m\,@\,\mathcal{I}'$, Lemma 4.8 is again the key: If χ is the first index with $p+1$ non-m-elements left of it, there are exactly $\chi - (p+1)$ m-type elements left of it! By the same arguments as above, all of those m-elements are located at positions in $\mathcal{I}'$, so we have $m\,@\,\mathcal{I}' = \chi - p - 1$ and by linearity of $\mathbb{E}$

$$\begin{aligned}\mathbb{E}\left[m\,@\,\mathcal{I}' \,|\, p, q\right] &= \mathbb{E}\,[\chi \mid p, q] - p - 1\\ &\underset{\text{Proposition 4.9}}{=} p + 1 + (q-p-1)\frac{p}{p+n-q} \quad - \quad p - 1\\ &= (q-p-1)\frac{p}{p+n-q}\,. \end{aligned}$$

Together, we have

$$\mathbb{E}\,[c_2 \mid p, q] = \frac{(p-1)^2}{n+p-q-1} + \frac{(q-p-1)p}{n+p-q}\,.$$

The usual application of total expectation gives

$$\begin{aligned}\mathbb{E}\, c_2 &= \mathbb{E}\left[\frac{(p-1)^2}{n+p-q-1}\right] + \mathbb{E}\left[\frac{(q-p-1)p}{n+p-q}\right] \\ &= \tfrac{2}{n(n-1)} \sum_{1\leqslant p<q\leqslant n} \tfrac{(p-1)^2}{n+p-q-1} \;+\; \tfrac{1}{6}(n+1) - \tfrac{1}{n}\,,\end{aligned}$$

where the second expectation has already been computed as $\mathbb{E}\,\mu$ in the proof of Proposition 4.9. Consider the remaining sum in isolation:

$$\begin{aligned}\tfrac{2}{n(n-1)} \sum_{1\leqslant p<q\leqslant n} \tfrac{(p-1)^2}{n+p-q-1} &= \tfrac{2}{n(n-1)} \sum_{p=2}^{n-1} (p-1)^2 \sum_{q=p+1}^{n} \tfrac{1}{n+p-q-1} \\ &= \tfrac{2}{n(n-1)} \sum_{p=1}^{n-2} p^2 \big(\mathcal{H}_{n-2} - \mathcal{H}_{p-1}\big) \\ &= \tfrac{2}{n(n-1)} \mathcal{H}_{n-2} \sum_{p=1}^{n-2} p^2 \;-\; \sum_{p=0}^{n-3} (p+1)^2 \mathcal{H}_p \\ &= \frac{2(n-2)(n-1)(2n-3)}{6n(n-1)} \mathcal{H}_{n-2} \;-\; \sum_{p=0}^{n-3} \Big(2\tbinom{p}{2} + 3p + 1\Big) \mathcal{H}_p \\ &= \tfrac{2}{9}n - \tfrac{5}{18} - \tfrac{1}{3}\tfrac{1}{n(n-1)}\,,\end{aligned}$$

where the last step splits the sum to apply eq. $(\Sigma i \mathcal{H}_i)$ on page 15 for $m = 0, 1, 2$. The lengthy terms are simplified by computer algebra.

Finally, we can put the two summands together and obtain

$$\begin{aligned}\mathbb{E}\, c_2 &= \Big(\tfrac{2}{9}n - \tfrac{5}{18} - \tfrac{1}{3}\tfrac{1}{n(n-1)}\Big) + \Big(\tfrac{1}{6}(n+1) - \tfrac{1}{n}\Big) \\ &= \tfrac{7}{18}n - \tfrac{1}{9} - \tfrac{1}{n} - \tfrac{1}{3}\tfrac{1}{n(n-1)}\,. \end{aligned} \tag{4.32}$$

4.3.4.5 c_3 in Algorithm 7

Frequency c_3 counts the occurrences of the comparison in line 16, which constitutes the loop condition of the second inner loop. Completely analogous to the argument for c_1 from Section 4.3.4.3, line 16 is executed once for every value of j except for the initialization value *right* $= n$. So at line 16, j attains the values

$$\mathcal{J} := \{n-1, n-2, \ldots, \chi\}\,. \tag{4.33}$$

(Note that j is *de*cremented.)
So, $c_3 = |\mathcal{J}| = n - \chi$ and by Proposition 4.9, we find

$$\begin{aligned}\mathbb{E}\,[c_3 \mid p, q] &= n - \mathbb{E}\,[\chi \mid p, q] \\ &= n - 1 - p - \frac{(q-p-1)p}{p+n-q} \\ \mathbb{E}\, c_3 &= n - \mathbb{E}\,\chi \\ &= \tfrac{1}{2}n - \tfrac{3}{2} + \tfrac{1}{n}\,. \end{aligned} \tag{4.34}$$

4.3.4.6 c_4 in Algorithm 7

Frequency c_4 counts how often line 17 is reached. Line 17 is inside the second inner loop, so c_4 behaves similar to c_2. However, there is a slight asymmetry to take into account: Whereas in the i-loop, we first check in line 9 for $i \geqslant j$, the j-loop first does the comparison in line 17 and then the check for $i \geqslant j$ in line 20. In fact, line 17 is reached for every value of j from $\mathcal{J}$ where $A[j] = A_0[j] \geqslant p$ (Lemma 4.7), so $c_4 = m@\mathcal{J} + l@\mathcal{J}$.

Both $\mathcal{J}$ and $c@\mathcal{J}$ are random and they are interdependent. Yet, we can compute the expectation similarly as in Section 4.3.4.4. We consider the two summands separately. For $l@\mathcal{J}$, we restrict our view to non-m-type positions: In Section 4.3.4.4, I show that there are in total $|S \cup L| = (p-1) + (n-q)$ such positions, $p-1$ of which are contained in $\mathcal{I}' = \{2, \dots, \chi - 1\}$. Now, $\mathcal{J} = \{2, \dots, n-1\} \setminus \mathcal{I}'$ is the complement of $\mathcal{I}'$ w. r. t. the non-pivot positions. Thus, $n-q$ non-m-positions are contained in $\mathcal{J}$ and eq. (4.13) yields

$$\mathbb{E}\,[l@\mathcal{J} \,|\, p, q] = (n-q)\frac{n-q}{(p-q)+(n-q)} \,.$$

For $m@\mathcal{J}$ we can also reuse knowledge from Section 4.3.4.4: $\mathbb{E}\,[m@\mathcal{I}' \,|\, p, q] = \frac{(q-p-1)p}{p+n-q}$ and by $\mathcal{J} = \{2, \dots, n-1\} \setminus \mathcal{I}'$ this gives

$$\mathbb{E}\,[m@\mathcal{J} \,|\, p, q] = (q-p-1) \; - \; \frac{(q-p-1)p}{p+n-q} \,.$$

Adding up already yields the conditional expected value for c_4. By symmetry $\sum_{1 \leqslant p<q \leqslant n} \frac{(p-1)^2}{(p-1)+(n-q)} = \sum_{1 \leqslant p<q \leqslant n} \frac{(n-q)^2}{(p-1)+(n-q)}$, so we can reuse the computations form Section 4.3.4.4 and find

$$\begin{aligned} \mathbb{E}\,[c_4 \mid p, q] &= \frac{(n-q)^2}{n+p-q-1} \; + \; (q-p-1) \; - \; \frac{(q-p-1)p}{n+p-q} \,. \\ \mathbb{E}\, c_4 &= \tfrac{2}{9}n - \tfrac{5}{18} - \tfrac{1}{3}\tfrac{1}{n(n-1)} \; + \; \tfrac{1}{3}(n-2) \; - \; \tfrac{1}{6}(n+1) + \tfrac{1}{n} \\ &= \tfrac{7}{18}n - \tfrac{10}{9} + \tfrac{1}{n} - \tfrac{1}{3}\tfrac{1}{n(n-1)} \,. \end{aligned} \tag{4.35}$$

4.3.4.7 s_0 in Algorithm 7

Line 5 is executed once per partitioning step where the pivots need to be swapped, so $\mathbb{E}\, s_0 = \frac{1}{2}$.

4.3.4.8 s_1 in Algorithm 7

Counter s_1 corresponds to line 11. This line is guarded by the comparison "A[i] < p" of line 10, so line 11 is executed for every value of i where $A_0[i] < p$: $s_1 = s@\mathcal{I}'$ with $\mathcal{I}' = \{2, \dots, \chi - 1\}$. Most fortunately, we already computed the expected value of $s@\mathcal{I}'$ in Section 4.3.4.4.

$$\begin{aligned} \mathbb{E}\,[s_1 \mid p, q] &= \frac{(p-1)^2}{(p-1)+(n-q)} \\ \mathbb{E}\, s_1 &= \tfrac{2}{9}n - \tfrac{5}{18} - \tfrac{1}{3}\tfrac{1}{n(n-1)} \end{aligned} \tag{4.36}$$

4.3.4.9 s_2 in Algorithm 7

The frequency s_2 belongs to line 18, which is the body of the if-statement inside the j-loop. In analogy to Section 4.3.4.8, line 18 is executed for every value of j with $A_0[j] > q$. i. e. $s_2 = l @ \mathcal{J}$. For the expected values, we can happily reuse calculations from Section 4.3.4.6:

$$\begin{aligned} \mathbb{E}\,[s_2 \mid p, q] &= \tfrac{(n-q)^2}{(p-1)+(n-q)} \\ \mathbb{E}\, s_2 &= \tfrac{2}{9}n - \tfrac{5}{18} - \tfrac{1}{3}\tfrac{1}{n(n-1)} = \mathbb{E}\, s_1 \;. \end{aligned} \tag{4.37}$$

4.3.4.10 s_3 in Algorithm 7

Frequency s_3 denotes the number of times line 23 is reached — which equals the number of outer loop iterations. To determine this frequency, consider the pointer i_1. It is only incremented during the swaps in lines 11 and 23. Moreover it is initialized to *left* $= 1$ and stops with $i_1 = p$, the rank of the small pivot. This means it is incremented $p-1$ times in total an thus $s_1 + s_3 = p - 1$. In Section 4.3.4.8, we found $s_1 = s @ \mathcal{I}'$ and computed its expected value. By linearity of the expectation, we close with

$$\begin{aligned} \mathbb{E}\,[s_3 \mid p, q] &= p - 1 - \mathbb{E}\,[s_1 \mid p, q] \\ &= p - 1 - \tfrac{(p-1)^2}{(p-1)+(n-q)} \;, \\ \mathbb{E}\, s_3 &= \mathbb{E}\, p - 1 - \mathbb{E}\, s_1 \\ &\underset{(\mathbb{E}p),(4.36)}{=} \tfrac{1}{3}(n-2) - \left(\tfrac{2}{9}n - \tfrac{5}{18} - \tfrac{1}{3}\tfrac{1}{n(n-1)}\right) \\ &= \tfrac{1}{9}n - \tfrac{7}{18} + \tfrac{1}{3}\tfrac{1}{n(n-1)} \;. \end{aligned} \tag{4.38}$$

4.3.4.11 s_4 and s_5 in Algorithm 7

The markers s_4 and s_5 correspond to lines 27 and 28. These lines consist of a plain array write operation, so one might wonder why I count them as a swap. However, line 4 contains the corresponding read operations. There is just some delay between the reads and the writes. As mentioned at the beginning of this section, Algorithm 7 does not use explicit exchanges, but rather moves a 'whole' through the array, which might be slightly cheaper, since we do not temporary storage. Nevertheless, we count lines 27 and 28 as one swap each. Of course, lines 27 and 28 are both executed exactly once per partitioning step, so

$$\begin{aligned} \mathbb{E}\, s_4 &= 1 \;, \\ \mathbb{E}\, s_5 &= 1 \;. \end{aligned}$$

4.3.5 Re-occurring Sums

In this section, I collect some sums that keep occurring as building blocks in the analyses in the preceding sections. All sums are straight-forward to evaluate and the closed forms are even found automatically by contemporary computer algebra systems.

$$\sum_{1\leqslant p<q\leqslant n} c = c\binom{n}{2} \tag{Σc}$$

$$\frac{2}{n(n-1)} \sum_{1\leqslant p<q\leqslant n} c = c \tag{$\mathbb{E}c$}$$

$$\sum_{1\leqslant p<q\leqslant n} p = \tfrac{1}{6}n(n^2-1) \tag{Σp}$$

$$\frac{2}{n(n-1)} \sum_{1\leqslant p<q\leqslant n} p = \tfrac{1}{3}(n+1) \tag{$\mathbb{E}p$}$$

$$\sum_{1\leqslant p<q\leqslant n} q = \tfrac{1}{3}n(n^2-1) \tag{Σq}$$

$$\frac{2}{n(n-1)} \sum_{1\leqslant p<q\leqslant n} q = \tfrac{2}{3}(n+1) \tag{$\mathbb{E}q$}$$

$$\sum_{1\leqslant p<q\leqslant n} p^2 = \tfrac{1}{12}n^2(n^2-1) \tag{Σp^2}$$

$$\sum_{1\leqslant p<q\leqslant n} q^2 = \tfrac{1}{12}(n-1)n(n+1)(3n+2) \tag{Σq^2}$$

$$\sum_{1\leqslant p<q\leqslant n} p\,q = \tfrac{1}{24}(n-1)n(n+1)(3n+2) \tag{Σpq}$$

$$\frac{2}{n(n-1)} \sum_{1\leqslant p<q\leqslant n} p\,q = \tfrac{1}{12}(n+1)(3n+2) \tag{$\mathbb{E}pq$}$$

4.4 Results & Discussion

In this chapter, we conducted a precise average case analysis of Algorithms 7 and 8 at the elementary operations level: Assuming the random permutation model, we computed the exact expected numbers of swaps and comparisons needed by the dual pivot Quicksort variants to sort a random list of n distinct elements. The results are shown in Tables 3 and 4. Plots of the total expected numbers of comparisons and swaps are shown in Chapter 6.

The analysis consisted of two parts: First in Section 4.2, we solved the dual pivot Quicksort recurrence of expected costs for general partitioning costs. Then we computed precise expected numbers of comparisons and swaps in the first partitioning step of Algorithms 7 and 8 in Section 4.3. Putting both together yields the results in Tables 3 and 4. For Algorithm 8, we can directly apply eq. (4.4) on page 61, for the solutions in Table 4 we have to start with eq. (4.2) on page 59.

location	exact expected executions	asymptotics (error $\mathcal{O}(\log n)$)
$\mathtt{C}_0$	$\frac{2}{5}n - \frac{1}{10}$	$0.4n$
$\mathtt{C}_1$	$\frac{4}{5}n\mathcal{H}_n - \frac{83}{50}n + \frac{4}{5}\mathcal{H}_n - \frac{2}{75}$	$0.8n \ln n - 1.198n$
$\mathtt{C}_2$	$\frac{1}{2}n\mathcal{H}_n - \frac{41}{40}n + \frac{1}{2}\mathcal{H}_n - \frac{1}{40}$	$0.5n \ln n - 0.736n$
$\mathtt{C}_3$	$\frac{2}{5}n\mathcal{H}_n - \frac{22}{25}n + \frac{2}{5}\mathcal{H}_n + \frac{1}{50}$	$0.4n \ln n - 0.649n$
$\mathtt{C}_4$	$\frac{1}{5}n\mathcal{H}_n - \frac{39}{100}n + \frac{1}{5}\mathcal{H}_n - \frac{7}{300}$	$0.2n \ln n - 0.275n$
$\mathtt{S}_0$	$\frac{1}{5}n - \frac{1}{20}$	$0.2n$
$\mathtt{S}_1$	$\frac{3}{10}n\mathcal{H}_n - \frac{127}{200}n + \frac{3}{10}\mathcal{H}_n - \frac{1}{600}$	$0.3n \ln n - 0.462n$
$\mathtt{S}_2$	$\frac{1}{5}n\mathcal{H}_n - \frac{39}{100}n + \frac{1}{5}\mathcal{H}_n - \frac{7}{300}$	$0.2n \ln n - 0.275n$
$\mathtt{S}_3$	$\frac{1}{10}n\mathcal{H}_n - \frac{49}{200}n + \frac{1}{10}\mathcal{H}_n + \frac{13}{600}$	$0.1n \ln n - 0.187n$
$\mathtt{S}_4 = \mathtt{S}_5$	$\frac{2}{5}n - \frac{1}{10}$	$0.4n$
$\mathtt{C} = \sum \mathtt{C}_i$	$\frac{19}{10}n\mathcal{H}_n - \frac{711}{200}n + \frac{19}{10}\mathcal{H}_n - \frac{31}{200}$	$1.9n \ln n - 2.458n$
$\mathtt{S} = \sum \mathtt{S}_i$	$\frac{3}{5}n\mathcal{H}_n - \frac{27}{100}n + \frac{3}{5}\mathcal{H}_n - \frac{19}{75}$	$0.6n \ln n + 0.076n$

Table 3: Total expected frequencies of all swap and comparison locations for dual pivot Quicksort with YAROSLAVSKIY's partitioning. The formulæ are obtained by inserting the results from Section 4.3.3 into eq. (4.4) on page 61.

location	exact expected executions	asymptotics (error $\mathcal{O}(\log n)$)
$\mathtt{C}_0$	$\frac{2}{5}n - \frac{1}{10}$	$0.4n$
$\mathtt{C}_1$	$\frac{3}{5}n\mathcal{H}_n - \frac{41}{50}n + \frac{3}{5}\mathcal{H}_n - \frac{11}{50}$	$0.6n\ln n - 0.474n$
$\mathtt{C}_2$	$\frac{7}{15}n\mathcal{H}_n - \frac{397}{450}n + \frac{7}{15}\mathcal{H}_n - \frac{149}{900}$	$0.4\overline{6}n\ln n - 0.613n$
$\mathtt{C}_3$	$\frac{3}{5}n\mathcal{H}_n - \frac{71}{50}n + \frac{3}{5}\mathcal{H}_n + \frac{9}{50}$	$0.6n\ln n - 1.074n$
$\mathtt{C}_4$	$\frac{7}{15}n\mathcal{H}_n - \frac{487}{450}n + \frac{7}{15}\mathcal{H}_n + \frac{121}{900}$	$0.4\overline{6}n\ln n - 0.813n$
$\mathtt{S}_0$	$\frac{1}{5}n - \frac{1}{20}$	$0.2n$
$\mathtt{S}_1$	$\frac{4}{15}n\mathcal{H}_n - \frac{122}{225}n + \frac{4}{15}\mathcal{H}_n - \frac{23}{900}$	$0.2\overline{6}n\ln n - 0.388n$
$\mathtt{S}_2$	$\frac{4}{15}n\mathcal{H}_n - \frac{122}{225}n + \frac{4}{15}\mathcal{H}_n - \frac{23}{900}$	$0.2\overline{6}n\ln n - 0.388n$
$\mathtt{S}_3$	$\frac{2}{15}n\mathcal{H}_n - \frac{76}{225}n + \frac{2}{15}\mathcal{H}_n + \frac{41}{900}$	$0.1\overline{3}n\ln n - 0.261n$
$\mathtt{S}_4 = \mathtt{S}_5$	$\frac{2}{5}n - \frac{1}{10}$	$0.4n$
$\mathtt{C} = \sum \mathtt{C}_i$	$\frac{32}{15}n\mathcal{H}_n - \frac{856}{225}n + \frac{32}{15}\mathcal{H}_n - \frac{77}{450}$	$2.1\overline{3}n\ln n - 2.573n$
$\mathtt{S} = \sum \mathtt{S}_i$[a]	$\frac{4}{5}n\mathcal{H}_n - \frac{19}{25}n + \frac{4}{5}\mathcal{H}_n - \frac{21}{100}$	$0.8n\ln n - 0.298n$

[a] Note that the line corresponding to $\mathtt{S}_3$ contributes two swaps, so $\mathtt{S} = \mathtt{S}_0 + \mathtt{S}_1 + \mathtt{S}_2 + 2\mathtt{S}_3 + \mathtt{S}_4 + \mathtt{S}_5$.

Table 4: Total expected frequencies of all swap and comparison locations for dual pivot Quicksort with SEDGEWICK's partitioning.

As most of the partitioning costs derived in Section 4.3.4 involve non-linear terms, we cannot directly apply eq. (4.4) on page 61. However, by linearity of the general solution eq. (4.2) on page 59, we can use eq. (4.4) to determine the contribution of the linear part of partitioning costs and then add the contribution of non-linear terms. The latter is derived directly from eq. (4.2) — in fact it suffices to compute two contributions:

(a) $pc_n = [n \geqslant 3] \cdot \frac{1}{n}$, which yields $C_n = \frac{1}{20}(n+1)$.

(b) $pc_n = [n \geqslant 3] \cdot \frac{1}{n(n-1)}$, which contributes $C_n = \frac{1}{60}(n+1)$.

For convenience, here are the exact coefficients of the leading $n \ln n$-term, again. The corresponding numbers for classic Quicksort from Section 3.3.2 on page 28 are also given.

	Comparison ($n \ln n + \mathcal{O}(n)$)	Swaps ($n \ln n + \mathcal{O}(n)$)
Classic Quicksort	2	$0.\overline{3}$
Dual Pivot Sedgewick	$2.1\overline{3}$	0.8
Dual Pivot Yaroslavskiy	1.9	0.6

In terms of comparisons, the new dual pivot Quicksort by Yaroslavskiy is best for large n. However, it needs more swaps, so whether it can outperform the classic Quicksort, depends on the relative runtime contribution of swaps and comparisons, which in turn differ from machine to machine. In Chapter 7, I will approach these relative contributions on the machine instruction level and Chapter 8 looks at wall clock running times in one particular setup. Remarkably, the new algorithm is significantly better than Sedgewick's dual pivot Quicksort in both measures. Given that Algorithms 7 and 8 are based on the same algorithmic idea, the considerable difference in costs is surprising. The explanation of the superiority of Yaroslavskiy's partitioning scheme is the major discovery of this chapter.

4.4.1 The Superiority of Yaroslavskiy's Partitioning Method

We proved that Yaroslavskiy's dual pivot Quicksort needs less comparisons than the classic one-pivot Quicksort and as Sedgewick's dual pivot variant. While the rigorous analysis was fun and all, one might easily lose the overall picture while figuring out special contribution $\delta = \sqrt{42}$ for comparison counter c_{147} ...

Just kidding. Seriously though, it pays to adopt a slightly more abstract view on the results. Would you have expected Algorithm 8 to save 5 % of all comparisons Algorithm 1 does? When I found out this difference, I immediately suspected such a pronounced effect to have an intuitive explanation, one that can be understood without digging into every detail of the implementation. And in fact, there is such an explanation.

However, before I explain how Yaroslavskiy's algorithm saves comparisons compared to classic Quicksort, let us discuss a — flawed — argument why this is impossible.

A Wrong Lower Bound for Dual Pivot Quicksort

Let $p < q$ be the two pivots. For partitioning, we need to determine for every $x \notin \{p, q\}$ whether $x < p$, $p < x < q$ or $q < x$ holds by comparing x to p *and/or* q. Assume, we first compare x to p, then averaging over all possible values for p, q and x, there is a $1/3$ chance that $x < p$ – in which case we are done. Otherwise, we still need to compare x and q. The expected number of comparisons for one element is therefore $1/3 \cdot 1 + 2/3 \cdot 2 = 5/3$. For a partitioning step with n elements including pivots p and q, this amounts to $5/3 \cdot (n-2)$ comparisons in expectation.

In the random permutation model, knowledge about an element $y \neq x$ does not tell us whether $x < p$, $p < x < q$ or $q < x$ holds. Hence, one could think that any partitioning method should need at least $5/3 \cdot (n-2)$ comparisons in expectation.
But this is not the case!

The argument seems quite plausible. In fact, previous work on multi-pivot Quicksort uses $5/3n + o(n)$ comparisons as partitioning cost for dual pivot Quicksort, e. g. [Hen91] and [Tan93]. This partitioning cost yields $2n \ln n + \mathcal{O}(n)$ comparisons in total — the same as classic Quicksort.

But where does the lower bound argument break down? The reason is the *independence* assumption above, which only holds true for algorithms that do comparisons at exactly *one location in the code* — like Algorithm 5. But Algorithms 7 and 8 have several compare-instructions at different locations, and how often those are reached *depends* on the pivots p and q. Now of course, the number of elements smaller, between and larger than p and q, directly depends on p and q, as well! So if a comparison is executed often if p is large, it is clever to first check $x < p$ there: The comparison is done more often than on average if and only if the probability for $x < p$ is larger than on average. Therefore, the expected number of comparisons can drop below the "lower bound" $5/3$ for this element!

And this is exactly, where Algorithms 7 and 8 differ: Yaroslavskiy's partitioning always evaluates the "better" comparison first, whereas in Sedgewick's dual pivot Quicksort this is not the case.

4.4.2 Making Sedgewick's Dual Pivot Quicksort Competitive

Now that we understand how the new Quicksort saves key comparisons, we can try to exploit our new knowledge for algorithmic improvements. In fact, there is no inherent reason to first compare elements with q in line 8 respectively with p in line 16 of Algorithm 7. We can simply *reverse* Sedgewick's order of comparisons to obtain a variant of Algorithm 7, which I will refer to as **Kciwegdes**. In the following chapter, I put this reversal idea to practice and analyze its impact.

5 Kciwegdes — Sedgewick Reversed

> *The Quicksort algorithm is better understood through analysis, and the analysis is very interesting in its own right. The many variations of the algorithm lead to much more spectacular variations in the analysis, and it is this combination of algorithm and analysis that makes the study of Quicksort so fascinating.*
>
> — R. Sedgewick [Sed75, page 265]

The analysis of Chapter 4 reveals that Sedgewick's partitioning (Algorithm 7) does more comparisons than necessary. In fact, the bottom line of Section 4.4.2 was that we can make Sedgewick's partitioning method competitive by *reversing* the order of comparisons in the inner loops. Algorithm 9 makes this idea concrete. To assess the effect of the reversal, I transfer the average case analysis of Chapter 4 to Algorithm 9.

Note that I also made the loop conditions strict before reversing the comparisons. This decent change was suggested in Section 4.1.1 because it dramatically improves performance in the presence of *equal elements*. For the analysis in this chapter, we will assume distinct elements. For that case, the change has no consequences at all.

5.1 Average Case Analysis

Luckily, much of the reasoning done for Algorithm 7 remains valid for Algorithm 9: Lemma 4.7 and Lemma 4.8 can be directly transferred, so we can make use of them in our analysis. Accordingly, Proposition 4.9 is available, as well. Moreover, reversing the order in which comparisons are done does not affect how often we swap, so all swap location frequencies are the same as for the original version of Sedgewick's partitioning method.

It remains to analyze the frequencies of the comparison markers. As intended, they behave differently for Algorithm 9. Yet, their analysis is still rather similar to the analyses done in Section 4.3.4 and we will only link to corresponding sections there, if arguments can be transferred.

Algorithm 9. Dual Pivot Quicksort with Kciwegdes Partitioning

```
DualPivotQuicksortKciwegdes($A$, left, right)
        // Sort the array $A$ in index range left, ..., right.
        if $right - left \geqslant M$          // Skip small subfiles ($M \geqslant 0$ a constant)
                $i := left$;        $i_1 := left$
                $j := right$;       $j_1 := right$
                $p := A[left]$;     $q := A[right]$
$c_0, [s_0]$        if $p > q$ then Swap $p$ and $q$ end if
                while true
                    $i := i + 1$
                    while true
                        if $i \geqslant j$ then break outer while end if
$c_1$                       if $A[i] < p$
$s_1$                           $A[i_1] := A[i]$; $i_1 := i_1 + 1$; $A[i] := A[i_1]$
$c_2$                       else if $A[i] \geqslant q$ then break inner while end if
                        $i := i + 1$
                    end while
                    $j := j - 1$
                    while true
$c_3$                       if $A[j] > q$
$s_2$                           $A[j_1] := A[j]$; $j_1 := j_1 - 1$; $A[j] := A[j_1]$
$c_4$                       else if $A[j] \leqslant p$ then break inner while end if
                        if $i \geqslant j$ then break outer while end if
                        $j := j - 1$
                    end while
$s_3$                   $A[i_1] := A[j]$; $A[j_1] := A[i]$
|                   $i_1 := i_1 + 1$;  $j_1 := j_1 - 1$
|                   $A[i] := A[i_1]$; $A[j] := A[j_1]$
                end while
$s_4$               $A[i_1] := p$
$s_5$               $A[j_1] := q$
                DualPivotQuicksortKciwegdes($A$,  left  , $i_1 - 1$)
                DualPivotQuicksortKciwegdes($A$, $i_1 + 1$, $j_1 - 1$)
                DualPivotQuicksortKciwegdes($A$, $j_1 + 1$,  right )
        end if
```

5.1.1 Who Crosses Whom

As we will see in the analysis, Algorithm 9 behaves slightly differently depending on which of the two crossing pointers i and j moves last. By the structure of Algorithm 9, i moves last *iff* we leave the outer loop via line 9. Let Φ be the indicator variable for that event, i. e.

$$\Phi := \begin{cases} 1 & \text{if the outer loop is left via line 9} \\ 0 & \text{if the outer loop is left via line 20} \end{cases} .$$

Lemma 5.1: $\Phi = 0$ iff $\mathrm{A}_0[\chi] > \mathrm{q}$ and conversely, $\Phi = 1$ iff $\mathrm{A}_0[\chi] < \mathrm{p} \vee \mathrm{q} = \mathrm{n}$.

Proof. It follows from Lemma 4.8, that $A_0[\chi] \in \mathrm{S} \cup \mathrm{L} \cup \{\mathrm{q}\}$. Since we assume distinct list elements, we have $\mathrm{A}_0[\chi] \in \{\mathrm{p},\mathrm{q}\}$ iff $\chi = \mathrm{n}$. Now if $\chi = \mathrm{n}$, we never took the break-branch in line 12. This implies $\mathrm{A}_0[i] < \mathrm{q}$ for $i \in \{2, \ldots, n-1\}$ and thus $\mathrm{q} = \mathrm{n}$. This shows that the right hand sides "$\mathrm{A}_0[\chi] > \mathrm{q}$" and "$\mathrm{A}_0[\chi] < \mathrm{p} \vee \mathrm{q} = \mathrm{n}$" of the two claimed equivalences are mutually exclusive. Thus, it suffices to prove both claimed implications from left to right.

To this end, assume $\Phi = 0$, i. e. the outer loop is left via line 20. To reach this point, we must have left the first inner loop via line 12, so we had $\mathrm{A}[\mathrm{i}] \geqslant \mathrm{q}$ there. As i has not been changed since then, $\mathrm{i} = \chi$ and by Lemma 4.7, $A_0[\chi] \geqslant \mathrm{q}$ follows. Additionally, as $\Phi = 0$, j has been decremented at least once, so $\chi < n$, such that $A_0[\chi] \neq \mathrm{q}$. It follows $A_0[\chi] > \mathrm{q}$, as claimed.

Now assume $\Phi = 1$. There are two cases here: Either we have left the first inner loop at least once — or never. In the latter case, $\chi = \mathrm{n}$ and we already argued above that $\mathrm{q} = \mathrm{n}$ in this case. So, suppose we have executed the j-loop at least once and consider now its last execution. This execution must have been quit via line 19 because of "$\mathrm{A}[\mathrm{j}] \leqslant \mathrm{p}$". j has not been changed since this last iteration and will never be changed again in this partitioning step. Hence, $\chi = \mathrm{j}$ and by Lemma 4.7 follows $\mathrm{A}_0[\chi] \leqslant \mathrm{p}$. As furthermore i is at least 2, also $\chi \geqslant 2$, such that we finally find $\mathrm{A}_0[\chi] < \mathrm{p}$. □

Proposition 5.2: For Φ holds

$$\mathbb{E}\,[\Phi \mid \mathrm{p},\mathrm{q}] = \begin{cases} 1 & \text{if } \mathrm{p} = 1 \text{ and } \mathrm{q} = \mathrm{n} \\ \frac{\mathrm{p}-1}{(\mathrm{p}-1)+(\mathrm{n}-\mathrm{q})} & \text{otherwise} \end{cases} ,$$

$$\mathbb{E}\,\Phi = \tfrac{1}{2} + \tfrac{1}{\mathrm{n}(\mathrm{n}-1)} . \tag{5.1}$$

Proof. Let us start with $\mathbb{E}\,[\Phi \mid \mathrm{p},\mathrm{q}]$ for the special case $\mathrm{q} = \mathrm{n}$. In this case, there cannot possible exist an index x, such that $\mathrm{A}_0[x] > \mathrm{q}$. In particular, $\mathrm{A}_0[\chi] \leqslant \mathrm{q}$, so by Lemma 5.1, $\Phi = 1$. Note that the claimed expression for $\mathbb{E}\,[\Phi \mid \mathrm{p},\mathrm{q}]$ is 1 whenever $\mathrm{q} = \mathrm{n}$, not only for $\mathrm{p} = 1$.

Now assume the other case $\mathrm{q} < \mathrm{n}$. By Lemma 5.1, $\mathrm{A}_0[\chi] \in \mathrm{S} \cup \mathrm{L}$ then. As all permutations are assumed equally likely, $\mathrm{A}_0[\chi]$ is uniformly distributed in $\mathrm{S} \cup \mathrm{L}$. Accordingly,

$$\Pr\bigl[\mathrm{A}_0[\chi] < \mathrm{p} \mid \text{pivots}\,\mathrm{p},\mathrm{q}\bigr] = \tfrac{\mathrm{p}-1}{(\mathrm{p}-1)+(\mathrm{n}-\mathrm{q})} .$$

Finally, again by Lemma 5.1, we have $\mathbb{E}\,[\Phi \mid \mathrm{p},\mathrm{q}] = \Pr\bigl[\mathrm{A}_0[\chi] < \mathrm{p} \mid \text{pivots}\,\mathrm{p},\mathrm{q}\bigr]$.

The unconditional expected value can now be computed as usual:

$$\begin{aligned}
\mathbb{E}\,\Phi &= \tfrac{2}{n(n-1)} \sum_{1\leqslant p<q\leqslant n} \mathbb{E}\,[\Phi \mid p, q] \\
&= \tfrac{2}{n(n-1)} \Biggl(\sum_{2\leqslant p<q\leqslant n} \tfrac{p-1}{p-1+n-q} + 1 \Biggr) \\
&= \tfrac{2}{n(n-1)} \sum_{q=3}^{n} \sum_{p=1}^{q-2} \Bigl(1 - \tfrac{n-q}{p+n-q} \Bigr) + \tfrac{2}{n(n-1)} \\
&= \tfrac{2}{n(n-1)} \sum_{q=0}^{n-3} \Biggl((n-q-2) - q \sum_{p=1}^{n-q-2} \tfrac{1}{p+q} \Biggr) + \tfrac{2}{n(n-1)} \\
&= \tfrac{2}{n(n-1)} \sum_{q=0}^{n-3} \Bigl((n-q-2) - q(\mathcal{H}_{n-2} - \mathcal{H}_q) \Bigr) + \tfrac{1+1}{n(n-1)} \\
&\underset{(\Sigma i \mathcal{H}_i)}{=} \frac{2(n-2)^2 - (n-3)(n-2)(\mathcal{H}_{n-2}+1) + 2\binom{n-2}{2}(\mathcal{H}_{n-2} - \frac{1}{2}) + 1}{n(n-1)} + \tfrac{1}{n(n-1)} \\
&= \frac{2(n-2)^2 - 3\binom{n-2}{2} + 1}{n(n-1)} + \tfrac{1}{n(n-1)} \,,
\end{aligned}$$

Now, factorizing the numerator yields $2(n-2)^2 - 3\binom{n-2}{2} + 1 = \frac{1}{2}n(n-1)$ and we conclude with $\mathbb{E}\,\Phi = \frac{1}{2} + \frac{1}{n(n-1)}$. □

5.1.2 c_0 in Algorithm 9

Line 5 corresponds to c_0 and is executed exactly once per partitioning step. Hence

$$c_0 = 1 \,. \tag{5.2}$$

5.1.3 c_1 in Algorithm 9

c_1 belongs to line 10. Its frequency is quite similar to that of line 8 of Algorithm 7 — however in Algorithm 9, the check for $i \geqslant j$ *precedes* this comparison. Therefore, we have two cases: If the outer loop if left via line 9, line 10 is *not* executed for the last value χ of i. On the other hand, if we leave the outer loop from line 20, we must have left the first inner loop through line 12. Then, line 10 has been executed for $i = \chi$. Using Φ from Section 5.1.1, we can write

$$\begin{aligned}
c_1 &= |\tilde{\mathcal{I}}| \,, \quad \text{where} \\
\tilde{\mathcal{I}} &:= \{2, \ldots, \chi - \Phi\} \,. \qquad\qquad (\tilde{\mathcal{I}} \subseteq \mathcal{I})
\end{aligned}$$

Then, $c_1 = |\tilde{\mathcal{J}}| = \chi - \Phi - 1$ and by using Proposition 4.9 and eq. (5.1) we find

$$\begin{aligned}
\mathbb{E}\,c_1 &= \mathbb{E}\,\chi - \mathbb{E}\,\Phi - 1 \\
&= \bigl(\tfrac{1}{2}n + \tfrac{3}{2} - \tfrac{1}{n}\bigr) - \bigl(\tfrac{1}{2} + \tfrac{1}{n(n-1)}\bigr) - 1 \\
&= \tfrac{1}{2}n - \tfrac{1}{n} - \tfrac{1}{n(n-1)} \,.
\end{aligned} \tag{5.3}$$

5.1.4 c_2 in Algorithm 9

Frequency c_2 counts executions of line 12, which is located in the else-branch of "$A[i] < p$" in line 10. In the then-branch of the same if-statement, we find line 11, i.e. the swap corresponding to s_1. Hence, $c_2 = c_1 - s_1$ and by linearity of $\mathbb{E}$

$$\begin{aligned}\mathbb{E}\, c_2 &= \mathbb{E}\, c_1 - \mathbb{E}\, s_1 \\ &\underset{(5.3),(4.36)}{=} \left(\tfrac{1}{2}n - \tfrac{1}{n} - \tfrac{1}{n(n-1)}\right) - \left(\tfrac{2}{9}n - \tfrac{5}{18} - \tfrac{1}{3}\tfrac{1}{n(n-1)}\right) \\ &= \tfrac{5}{18}n + \tfrac{5}{18} - \tfrac{1}{n} - \tfrac{2}{3}\tfrac{1}{n(n-1)} \,. \end{aligned} \tag{5.4}$$

5.1.5 c_3 in Algorithm 9

The location corresponding to c_3 in Algorithm 9 is line 17. It is executed exactly as often as line 16 of Algorithm 7, namely for every value of j in $\mathcal{J} = \{n-1, n-2, \ldots, \chi\}$. As discussed in Section 4.3.4.5, $c_3 = |\mathcal{J}|$ and

$$\mathbb{E}\, c_3 = \tfrac{1}{2}n - \tfrac{3}{2} + \tfrac{1}{n} \,. \tag{5.5}$$

5.1.6 c_4 in Algorithm 9

Finally, for line 19 — whose frequency is c_4 — the same argumentation as in Section 5.1.4 applies: After the comparison in line 17, we *either* do the swap in line 18 *or* we execute line 19. Accordingly, $c_4 = c_3 - s_2$.

$$\begin{aligned}\mathbb{E}\, c_4 &= \mathbb{E}\, c_3 - \mathbb{E}\, s_2 \\ &\underset{(5.5),(4.37)}{=} \left(\tfrac{1}{2}n - \tfrac{3}{2} + \tfrac{1}{n}\right) - \left(\tfrac{2}{9}n - \tfrac{5}{18} - \tfrac{1}{3}\tfrac{1}{n(n-1)}\right) \\ &= \tfrac{5}{18}n - \tfrac{11}{9} + \tfrac{1}{n} + \tfrac{1}{3}\tfrac{1}{n(n-1)} \,. \end{aligned} \tag{5.6}$$

5.2 Results & Discussion

In this chapter, we used the insight gained during the analysis of Algorithm 7 in Chapter 4 to create the reversed version of SEDGEWICK's partitioning method. The result is shown in Algorithm 9. Moreover, to evaluate whether the modification was an actual improvement, we performed an average case analysis in the flavor of Chapter 4 for Algorithm 9. The results are shown in Table 5.

It is evident that in terms of elementary operations, Algorithm 9 is superior to Algorithm 7. Actually, I find it very remarkable that such a small change in the algorithm suffices to save *one eighth of all comparisons* in the average.

Looking at our situation from a broader perspective, we now have three competitive algorithms:

- Classic Quicksort shines when it comes to the number of swaps.
- Dual Pivot Quicksort with YAROSLAVSKIY's partitioning method can save some comparisons, but needs more swaps.
- Dual Pivot Quicksort with KCIWEGDES partitioning needs even more swaps than YAROSLAVSKIY's method, but it achieves yet another reduction in the number of needed key comparisons.

Here is the updated summary table of leading term coefficients:

	Comparison ($n \ln n + \mathcal{O}(n)$)	Swaps ($n \ln n + \mathcal{O}(n)$)
Classic Quicksort	2	$0.\overline{3}$
Dual Pivot SEDGEWICK	$2.1\overline{3}$	0.8
Dual Pivot YAROSLAVSKIY	1.9	0.6
Dual Pivot KCIWEGDES	$1.8\overline{6}$	0.8

This leaves us with the question, which of these algorithms to use. Counting elementary operations cannot get us closer to the answer. Instead, it is time for a more fine-grained cost model! This is where the journey will lead us to in Chapter 7.

location	exact expected executions	asymptotics (error $\mathcal{O}(\log n)$)
$\mathtt{C}_0$	$\frac{2}{5}n - \frac{1}{10}$	$0.4n$
$\mathtt{C}_1$	$\frac{3}{5}n\mathcal{H}_n - \frac{163}{150}n + \frac{3}{5}\mathcal{H}_n - \frac{71}{300}$	$0.6n\ln n - 0.740n$
$\mathtt{C}_2$	$\frac{1}{3}n\mathcal{H}_n - \frac{49}{90}n + \frac{1}{3}\mathcal{H}_n - \frac{19}{90}$	$0.\overline{3}n\ln n - 0.352n$
$\mathtt{C}_3$	$\frac{3}{5}n\mathcal{H}_n - \frac{71}{50}n + \frac{3}{5}\mathcal{H}_n + \frac{9}{50}$	$0.6n\ln n - 1.074n$
$\mathtt{C}_4$	$\frac{1}{3}n\mathcal{H}_n - \frac{79}{90}n + \frac{1}{3}\mathcal{H}_n + \frac{37}{180}$	$0.\overline{3}n\ln n - 0.685n$
$\mathtt{S}_0$	$\frac{1}{5}n - \frac{1}{20}$	$0.2n$
$\mathtt{S}_1$	$\frac{4}{15}n\mathcal{H}_n - \frac{122}{225}n + \frac{4}{15}\mathcal{H}_n - \frac{23}{900}$	$0.2\overline{6}n\ln n - 0.388n$
$\mathtt{S}_2$	$\frac{4}{15}n\mathcal{H}_n - \frac{122}{225}n + \frac{4}{15}\mathcal{H}_n - \frac{23}{900}$	$0.2\overline{6}n\ln n - 0.388n$
$\mathtt{S}_3$	$\frac{2}{15}n\mathcal{H}_n - \frac{76}{225}n + \frac{2}{15}\mathcal{H}_n + \frac{41}{900}$	$0.1\overline{3}n\ln n - 0.261n$
$\mathtt{S}_4 = \mathtt{S}_5$	$\frac{2}{5}n - \frac{1}{10}$	$0.4n$
$\mathtt{C} = \sum \mathtt{C}_i$	$\frac{28}{15}n\mathcal{H}_n - \frac{794}{225}n + \frac{28}{15}\mathcal{H}_n - \frac{73}{450}$	$1.8\overline{6}n\ln n - 2.451n$
$\mathtt{S} = \sum \mathtt{S}_i$	$\frac{4}{5}n\mathcal{H}_n - \frac{19}{25}n + \frac{4}{5}\mathcal{H}_n - \frac{21}{100}$	$0.8n\ln n - 0.298n$

Table 5: Total expected frequencies of all swap and comparison locations for dual pivot Quicksort with KCIWEGDES partitioning. Note that the results for swaps are copied from Table 4.

6 Predictive Quality of the Expectation

> " *If the facts don't fit the theory, change the facts.* "
>
> — attributed to Albert Einstein

In Chapters 4 and 5, we obtained precise expected values for the number of swaps and comparisons used by the considered dual pivot Quicksort variants on a random permutation of size n. Whereas the average complexity is $\Theta(n \log n)$, all variants have quadratic worst case inputs. This might cast doubts on the predictive quality of the expectation: If there is a worst case which is so far away from the expected value, how much trust can be put in the average case? The short answer is: Quite much.

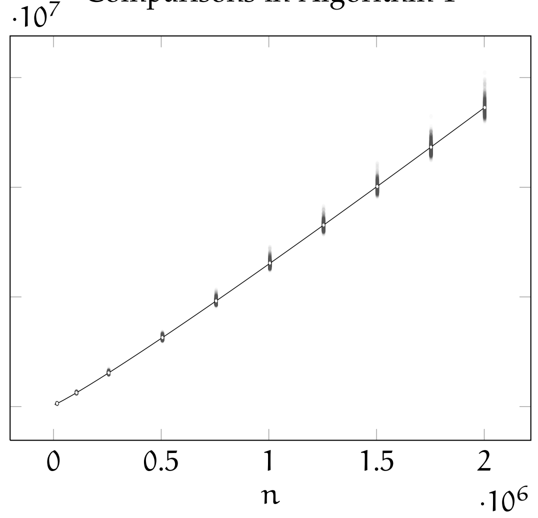
For the long answer, one should have a closer look at the *distribution of costs*. The arguably simplest parameter of a distribution allowing to assess the predictive power of the mean is the *standard deviation* of the distribution. Then, Chebyshev's inequality states that a $1 - \frac{1}{k^2}$ fraction of all inputs cause costs of $\mu \pm k \cdot \sigma$ where μ are the expected costs and σ is the standard deviation. If further $\sigma \in o(\mu)$ for $n \to \infty$, this means that the relative deviation from the mean $\frac{\mu \pm k \cdot \sigma}{\mu} \to 1$ for any constant k; differently stated: For any constant probability p and error ϵ, there is a n_0 such that for $n \geqslant n_0$, the probability of a relative deviation from the mean of more than ϵ is less than p.

For classic Quicksort with median of k, e. g. Hennequin shows in [Hen89, Section 4.3] that the variance of the number of comparisons and swaps is in $\Theta(n^2)$ for any constant k. In particular, this holds for $k = 1$ and thus for Algorithm 1. In [Hen91, Proposition IV.8], the result for comparisons is even generalized to Algorithm 5 with arbitrary constant s and t.

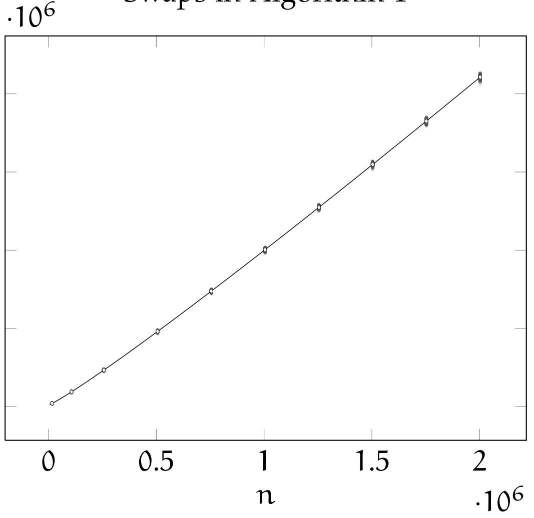
A crucial argument in Hennequin's derivation is that the number of comparisons used in the first partitioning step of Algorithm 5 only depends on the size of the input, but is stochastically *independent* of the list itself. Due to the asymmetry discovered in the preceding chapters, this is not the case for our dual pivot Quicksort variants. Therefore, precise analysis of variances is more difficult for Algorithms 8 and 9 and not considered in this thesis.

However, I empirically investigated by how much actual costs deviate from the expected value. The following series of plots shows the number of comparisons and swaps needed to sort the random lists generated according to the parameters of Section 8.1.2. For every input size, 1000 random permutations are used. The cost for sorting one of them contributes one semi-transparent gray blob in the plot. Where many points are close to each other, their opacity adds up, so the darkness of an area corresponds to its probability mass. The smaller white dot inside the gray lines shows the sample mean. Finally, the continuous line shows the expected values computed in the preceding chapters.

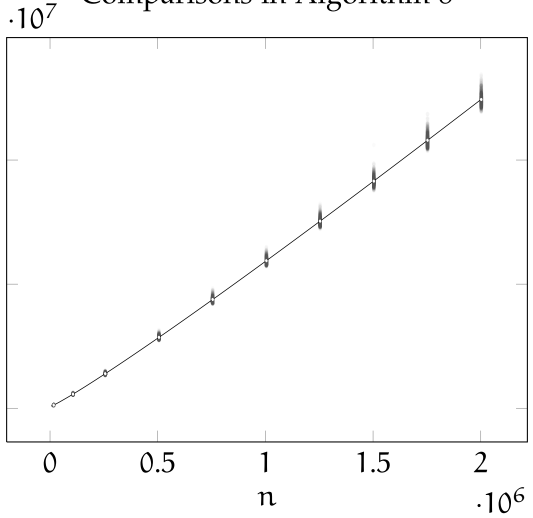
Comparisons in Algorithm 1



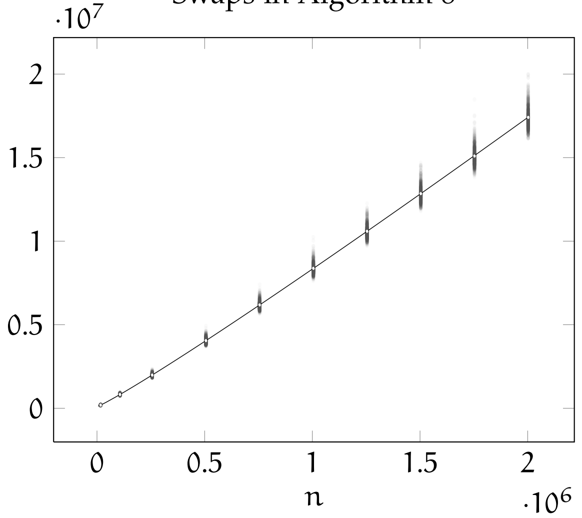
Swaps in Algorithm 1



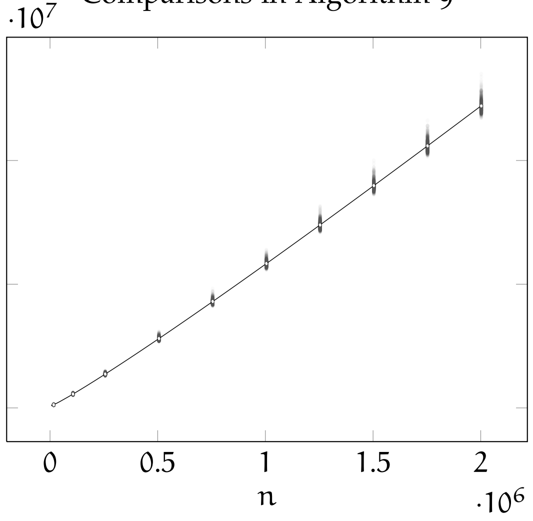
Comparisons in Algorithm 8



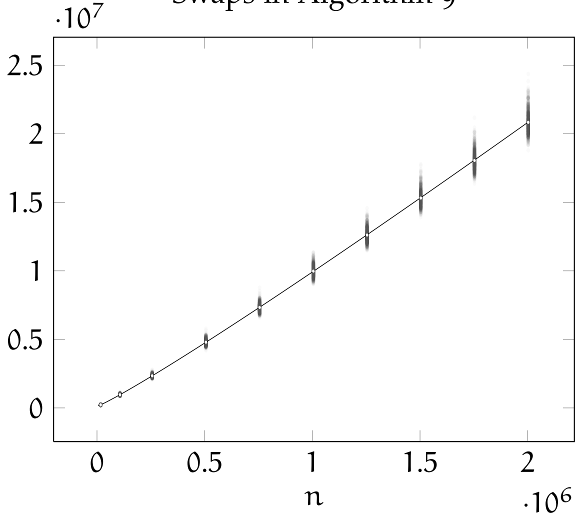
Swaps in Algorithm 8

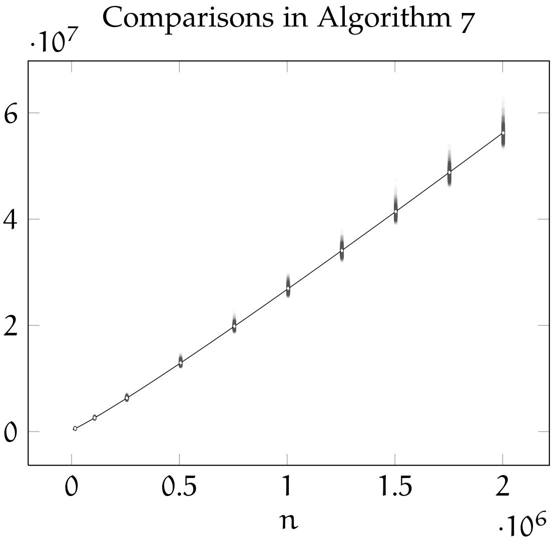


Comparisons in Algorithm 9

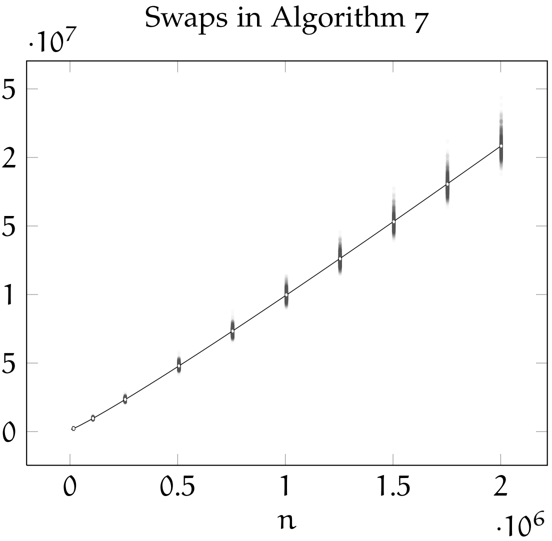


Swaps in Algorithm 9





Comparisons in Algorithm 7

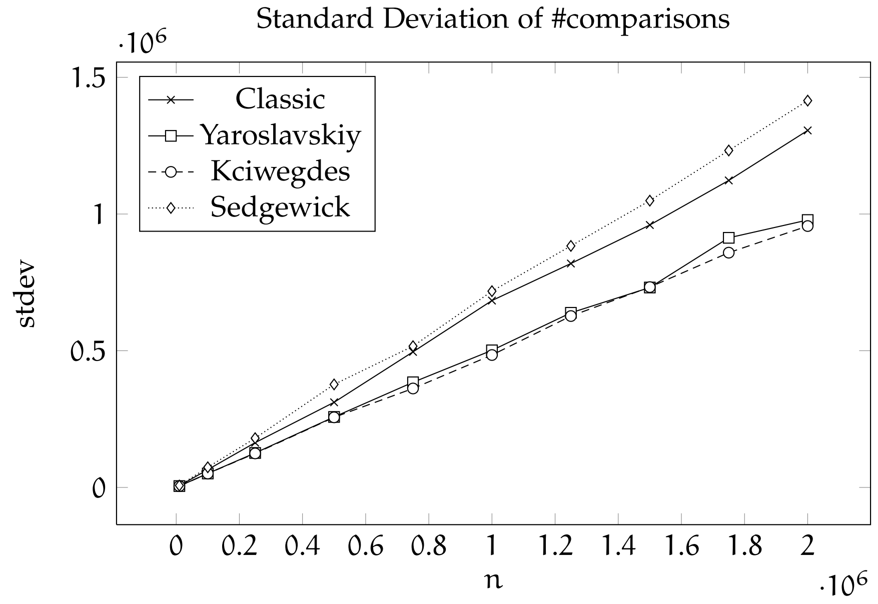


Swaps in Algorithm 7

It is remarkable that even for the moderate size of 1000, the sample means show no visible deviation from the expected value. This is also reassuring in the face of the many error-prone calculations involved in the derivation of the expected values.

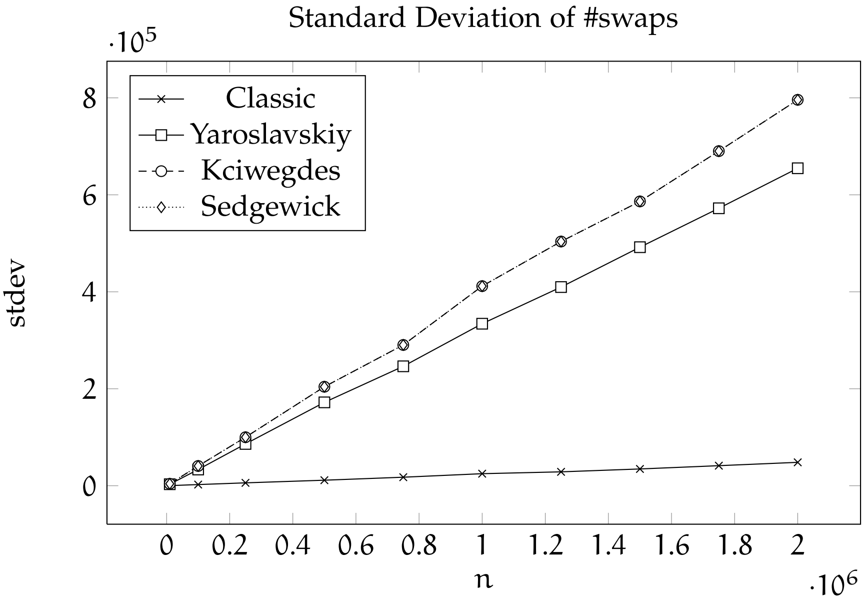
The asymmetry of the distribution of the number of comparisons is clearly visible: Deviations above the mean are more extreme than those below it. This fits known properties of the limiting law for classic Quicksort, e. g. [KS99]. An empirically determined plot of its density can be found e. g. in [Hen91, Figure IV.2].

In absolute terms, all measurements remain quite close to the mean. While it is evident in the above plots that the variance increases, it is hard to speculate at the order of growth. It is natural to hypothesize quadratic growth as for Algorithm 5, i. e. linear growth for the standard deviation. To test this hypothesis, here is a plot of the standard deviation of the samples for the different values of n.



Standard Deviation of #comparisons

Judging from this plot, linear growth is not implausible. In contrast to the number of comparisons, much less is known about the distribution of swaps, since their number is much more dependent on the algorithm considered. Below, I show the same plot as above for the number of swaps. Here as well, the standard deviation appears to grow linearly in n. The difference in variance between classic Quicksort and the dual pivot variants is remarkable. Note further that SEDGEWICK's and KCIWEGDES partitioning use *exactly* the same swaps, hence the variance points coincide, as well.



7 Counting Primitive Instructions

> *Not everything that can be counted counts, and not everything that counts can be counted.*
>
> — attributed to Albert Einstein

The study of algorithms by counting the number of dominating elementary operations — as done in Chapter 4 — is inherently limited. Here are some of the potential problems arising from such an approach.

- First of all, we have to somehow *define the elementary operations*. Sometimes, good candidates are already given by the problem definition. In fact, this is the case for the sorting problem definition from Section 2.2. However, it might be the case that an abstract definition does not explicitly list all relevant operations. Or, some of the "elementary operation" turn out not to be elementary after all, if they are not implementable in constant time.

- Analyzing the (expected) number of certain elementary operations only helps comparing algorithms using the same operations. For example, assume we know more about the elements we are sorting. Then, more efficient, specialized sorting methods are available, e. g. distribution counting [Knu98, Algorithm 5.2D]. This algorithm is based on *different elementary operations*, so an analysis in the flavor of Chapter 4 does not allow comparing it to our Quicksort variants.

- By looking only at elementary operations, we typically *lose lower terms* of the runtime. For example, in Quicksort, swaps and comparisons are the dominant operations — a linear combination of their frequency determines the leading $n \ln n$ term of the total runtime. For moderately sized inputs, however, the linear term in runtime yields non-negligible contributions. As we will see below, the linear term is mostly due to overhead per partitioning step. Only counting the overall number of swaps and comparisons does not capture this overhead.

- Most algorithms involve *several types of elementary operations*. Counting them separately often results in two algorithms to be *incomparable*. This is the case with Algorithms 1 and 8: Algorithm 1 needs less swaps, but more comparisons than Algorithm 8. Unless we somehow determine the relative runtime contributions of all operations, we cannot rank the algorithms.

The last point is the most important shortcoming of counting abstract operations in the author's opinion. It severely limits the usefulness of such analyses in choosing the best algorithm for a problem at hand.

Counting primitive instructions for an implementation on a particular machine provides a sound way to derive relative runtime contributions of elementary operations. The exact ratios certainly depend on the machine and details of the implementation, so they only allow to compare implementations, not abstract algorithms. Nevertheless, one should expect to find roughly similar ratios for 'similar' implementations on 'similar' machines, such that distinct qualitative rankings might be transferable.

Ambiguity Alarm

It should be noted for clarity that the term "relative runtime contribution" of elementary operations can be understood in two rather different ways: First, we can simply consider the computation time it takes to execute a single occurrence of an elementary operation in isolation. As different operations involve different computation steps, their computation time will differ. Several executions of the same operation type always have the same running time.

This model is somehow tacitly assumed in Chapter 4, where we added up all frequencies for all comparison locations to get the total number of comparisons. This total tally is only appropriate, if all comparisons are assumed to yield the *same* cost contribution. For predicting runtime, the mere number of an elementary operation is only interesting, if we assume the rest of the program to have negligible cost.

The second interpretation of "relative runtime contribution" is the one adopted in this chapter. We perceive the elementary operation locations as mere *markers* of certain basic blocks, and the contribution of one elementary operation is the cost of the *whole basic block* in which the operation location happens to reside. Then, different locations for the same operation may naturally yield very different runtime contributions.

Section 7.4.1 uses the implementations discussed in this chapter to derive the relative runtime impact of swaps and comparisons in our Quicksort variants.

The Devil is in the details.

There is a hidden subtlety in this second interpretation of relative runtime contributions: It may happen that a *single* basic block contains *more than one* elementary operation. Then, it is not clear, whom to assign the costs of this block. In fact, Algorithm 8 contains a block with both a swap *and* a comparison. We will propose and discuss a possible solution in Section 7.4.1.

The instruction counts obtained in this chapter are additive cost models in the sense of Section 2.5 on page 21. Hence, we can analyze the expected costs via KNUTHian analysis.

As our programs tend be somewhat lengthy, we use a trivial variation of the methodology. We first partition the program listing into **basic blocks**, i. e. maximal sequences of instructions which are *always* executed sequentially. By definition, every instruction in a block is executed the same number of times as any other instruction in the same block. Hence, we can replace a basic block by a imaginary instruction whose cost contribution is the sum of all contributions in the block. Then, we apply KNUTHian analysis as described in Section 2.5.

When it comes to implementations of algorithms, we have to decide on which *machine* the implementation is to be run. This choice is vital as it inevitably influences how we implement an algorithm. Furthermore, the machine partially dictates the cost model. I try to alleviate the severity of this decision by not choosing *one* machine, but *two* machines — at the price of double effort . . .

The first machine is the "mythical computer" MMIX which KNUTH uses in his books to describe and analyze algorithms. The main advantages over real machines is that the runtime behavior is well-defined and the architecture is free of legacy quirks. In practice, backward-compatibility often takes precedence over clean design. A mythical machine need not bother with that. That — and the many well-studied programs from KNUTH's books — made this machine the arguably canonical choice for theoretically inclined algorithm researchers.

As second platform I chose the Java Virtual Machine (JVM), which interprets Java Bytecode. The reason for my choice is twofold. First of all, YAROSLAVSKIY's algorithm originates from and was adopted by the Java community. Thus, it is natural to study the efficiency of an implementation in Java Bytecode. Second, the JVM has become one of the major software platforms over the last decade — both for running commercial software and for teaching algorithms (e. g. in [SW11]). Therefore, performance evaluations of Bytecode implementations are of immediate practical interest.

By T_n^{CM}, I denote the **expected cost for sorting a random permutation of size n in the cost model CM**. Accordingly, I write $T_n^{\mathtt{MMIX}}$ and T_n^{JVM} for the costs of the MMIX and JVM implementations, respectively.

7.1 MMIX

> *MMIX [...] is very much like nearly every general-purpose computer designed since 1985, except that it is, perhaps, nicer.* — D. E. Knuth in [Knu05]

For the first edition of his book series "The Art of Computer Programming", Knuth designed a "mythical machine" called MIX. His intention was to create a detailed model computer that can be used to study and compare implementations of algorithms. Therefore, the model should be similar to actual machines, which MIX achieved for over 20 years with flying colors. However, the advent of reduced instruction set computers (RISC) caused a fundamental change in processor architectures. This moved modern computers quite far away from MIX.

Knuth realized this change very early and designed a successor for MIX during the 1990's. The result is called MMIX and is presented in [Knu05]. MMIX gets rid of some intricacies[18] of MIX, while retaining its likable features. Most importantly, every instruction causes well-defined costs. The costs are expressed in υ ("oops") and μ ("mems"), where one υ represents one processor clock cycle and one μ symbolizes one memory access. The cost contribution for each instruction type can be found in Table 1 of [Knu05].

In this basic cost model, advanced pipelined execution and caching are neglected. An exception, however, is the inclusion of *static branch prediction*. Every conditional jump exists in two flavors: A standard version and a *probable* jump version. They only differ in their prediction which outcome of the conditional jump is to be expected: The standard branch expects not to jump, whereas the probable branch expects the jump to happen. MMIX will prepare to seamlessly follow the predicted route. However, if the prediction was wrong, these preparations need to be undone. The model accounts for that by charging 2υ to mispredicted branches.

This treatment of mispredicted branches uses context information of the trace and hence is not a constant additive cost model (recall the definitions from Section 2.5 on "Knuthian Analysis of Algorithms"). However, we can transform it into one! For every branch instruction, we add an imaginary instruction / basic block with weight 2υ 'inside' the edge for the mispredicted branch. This means, the imaginary instruction is inserted into the trace after a branch instruction iff that branch was mispredicted. The cost including branch mispredictions via this augmented traces is then a constant additive cost model. So, we can apply Knuthian analysis. To do so, we need to derive the frequency of the imaginary blocks. Using Kirchhoff's laws, we can compute the flow on all edges from the excesses of all nodes. So, the additional effort is quite small.

[18] "one couldn't even use it with ASCII code to print lowercase letters. And ouch, its standard subroutine calling convention was irrevocably based on self-modifying instructions!" (from the Preface of [Knu05]).

Listing 1. MMIX implementation of Classic Quicksort (Algorithm 1).

```
Qsort  ⌜CMP   tmp,lleft,rright      R       if left ⩾ right terminate.
       ⌞PBNN  tmp,9F                R
       ⌜LDO   p,A,rright            A       pivot p := A[right].
        SUBU  ii,lleft,8            A       i := left − 1.
       ⌞SET   jj,rright             A       j := right.
                                            do ...
1H     ⌜ADDU  ii,ii,8               C1      | do i := i + 1
        LDO   Ai,A,ii               C1      |
        CMP   tmp,Ai,p              C1      | while A[i] < p
       ⌞PBN   tmp,1B                C1      |
2H     ⌜SUBU  jj,jj,8               C2      | do j := j − 1
        LDO   Ai,A,jj               C2      |
        CMP   tmp,Ai,p              C2      | while A[j] > q
       ⌞PBP   tmp,2B                C2      |
       ⌜CMP   tmp,jj,ii             S1 + A  | if j > i
       ⌞BNP   tmp,after             S1 + A  | (i. e. jump away if j ⩽ i)
       ⌜STO   Ai,A,jj               S1      | | Swap A[i] and A[j].
        STO   Ai,A,ii               S1      | |
                                            while j > i
       ⌞JMP   1B                    S1      (We checked j > i above already.)
after  ⌜LDO   Ai,A,ii               A       Swap A[i] and pivot.
        STO   p,A,ii                A
        STO   Ai,A,rright           A
                                          Recursive Calls: Rescue needed registers.
        STO   rright,sp,1*8         A       (A will just stay in the register.)
        STO   return,sp,2*8         A
        STO   ii,sp,3*8             A
                                          Quicksort(A, left, i − 1)
                                            left already in arg1
        SUBU  arg2,ii,8             A       arg2
        ADDU  sp,sp,4*8             A       Advance stackpointer.
        GETA  argRet,@+8            A       Store return address.
       ⌞JMP   Qsort                 A       Jump back to start.
       ⌜SUBU  sp,sp,4*8             A       Pop stored registers from stack.
                                          Quicksort(A, i + 1, right)
        LDO   arg1,sp,3*8           A       Restore i.
        ADDU  arg1,arg1,8           A       arg1 := i + 1.
        LDO   arg2,sp,1*8           A       Restore right.
        ADDU  sp,sp,4*8             A       Advance stackpointer.
        GETA  argRet,@+8            A       Store return address.
       ⌞JMP   Qsort                 A       Jump back to start.
       ⌜SUBU  sp,sp,4*8             A       Pop stored registers from stack.
       ⌞LDO   return,sp,2*8         A       Only restore return here.
9H     ⌜GO    return,return,0       R       Return to caller.
```

Lines	Frequency	instruction costs	Mispredicted Branch	Frequency
1–2	R	2υ	$2 \rightsquigarrow 3$	A
3–5	A	$3\upsilon + 2\mu$		
7–10	C_1	$4\upsilon + \mu$	$10 \rightsquigarrow 11$	$S_1 + A$
11–14	C_2	$4\upsilon + \mu$	$14 \rightsquigarrow 15$	$S_1 + A$
15–16	$S_1 + A$	2υ	$16 \rightsquigarrow 21$	A
17–20	S_1	$3\upsilon + 2\mu$		
21–33	A	$11\upsilon + 7\mu$		
34–41	A	$7\upsilon + 2\mu$		
42–43	A	$2\upsilon + \mu$		
44–44	R	3υ		

Table 6: Basic Block Instruction costs for the MMIX implementation (Listing 1) of classic Quicksort (Algorithm 1). The costs are given in terms of υ ("oops") and μ ("mems"). Moreover, for every conditional jump, the frequency of the mispredicted branch is given.

Lines	Frequency	instruction costs	Mispredicted Branch	Frequency
1–2	R	2υ	$2 \rightsquigarrow 3$	A
3–6	A	$4\upsilon + 2\mu$	$6 \rightsquigarrow 10$	$A - S_0$
7–9	S_0	3υ		
10–12	A	3υ		
13–14	$C_1 + A$	2υ	$14 \rightsquigarrow 43$	A
15–17	C_1	$3\upsilon + \mu$	$17 \rightsquigarrow 18$	S_1
18–22	S_1	$5\upsilon + 3\mu$		
23–24	$C_1 - S_1$	2υ	$24 \rightsquigarrow 25$	S_2
25–27	C_3	$3\upsilon + \mu$	$27 \rightsquigarrow 32$	$C_3 - X$
28–29	X	2υ	$29 \rightsquigarrow 30$	$X - (C_3 - S_2)$
30–31	$C_3 - S_2$	2υ		
32–36	S_2	$5\upsilon + 2\mu$	$36 \rightsquigarrow 37$	S_3
37–40	S_3	$4\upsilon + 3\mu$		
41–42	C_1	2υ		
43–61	A	$17\upsilon + 11\mu$		
62–70	A	$8\upsilon + 2\mu$		
71–78	A	$7\upsilon + 2\mu$		
79–80	A	$2\upsilon + \mu$		
81–81	R	3υ		

Table 7: Basic Block Instruction costs for the MMIX implementation (Listing 2) of dual pivot Quicksort with YAROSLAVSKIY's partitioning (Algorithm 8). The costs are given in terms of υ ("oops") and μ ("mems"). Moreover, for every conditional jump, the frequency of the mispredicted branch is given.

7.1.1 Remarks for MMIX Implementations

As MMIX is a 64bit computer, it is natural to assume A to be the base address of an array of $n + 1$ 64-bit-integer numbers. The first number is the smallest representable integer, which we use as a sentinel. The following n numbers are the elements to be sorted.

Although the basic unit of operation for MMIX is an 8 byte word, the unit of main memory is addresses is single bytes. Therefore, the address of the ith entry of the array A is found at address $\mathtt{A} + 8i$. It is then more efficient to directly store $8i$ instead of i for all array indices. I will indicate this by doubling the first letter of a variable name, i. e. if the pseudocode variable is called *left* and we actually store $8 \cdot \mathit{left}$, I will call the corresponding register `lleft`. I have given all registers used in the MMIX implementations symbolic names. In the actual MMIXAL[19] programs, these symbolic names are mapped to actual register numbers.

A nice feature of MMIX is the direct support for procedure calls using the PUSHJ and POP instructions. However, these instructions hide some of the costs of the call stack management[20]. Therefore, I decided to implement explicit stack management manually. Register sp is the stack pointer, which always points to the topmost element on the stack. Then pushing a word onto the stack consists of writing it to address $\mathtt{sp} + 8$ and setting $\mathtt{sp} := \mathtt{sp} + 8$. Accordingly, we can pop the topmost word off the stack by reading it from address sp and setting $\mathtt{sp} := \mathtt{sp} - 8$.

7.1.2 MMIX Implementation for Algorithm 1

An MMIX implementation of Algorithm 1 is given in Listing 1. Registers `lleft` and `rright` contain *left* and *right* respectively and register A contains the base address of array A. i and j are stored in registers `ii` and `jj`, respectively.

Basic blocks are embraced by $\begin{smallmatrix}\ulcorner\\ \llcorner\end{smallmatrix}$ and for each line, its frequency is given after the instruction. Many of the frequencies are counters of swaps and comparisons markers, which were defined in Section 4.3.1 and whose expected values are given in Table 1 on page 30. For the remaining ones, we determine the expected value in Section 7.3.

Listing 1 contains four conditional jumps: The first in line 2 skips the whole partitioning step if needed. As Section 7.3.2 will show, it is actually slightly more probable that we skip the partitioning step, so we use the *probable jump* version of the branch instruction. Accordingly, line 2 causes a mispredicted branch for every real partitioning step, i. e. in total A mispredictions.

Similar reasoning applies to the other branch locations. lines 10 and 14 form the back-jump of the inner loops of Algorithm 1. The inner loops are left exactly as often as the block following the loops is executed, which is $\mathtt{S}_1 + A$. Finally, line 16 terminates the outer loop, which happens exactly A times in total.

[19] MMIXAL is the assembly languages for MMIX, described in [Knu05, Section $1.3.2'$].

[20] The costs for PUSHJ and POP are υ and and 3υ respectively, i. e. both do not involve memory accesses. However, since the number of registers is constant, but the call stack is unbounded, we eventually need to put part of the stack into memory.

Summing the products of cost contributions and frequencies — including those for the branch mispredictions — gives the following grand total

$$T_n^{\mathtt{MMIX}} = A(33\upsilon + 12\mu) + R \cdot 5\upsilon + (C_1 + C_2)(4\upsilon + \mu) + S_1(9\upsilon + 2\mu) \ .$$

Note that the two different comparison locations contribute by the same amount to the total cost. This is due to the symmetry of the inner loops. Therefore it suffices to know $C := C_1 + C_2$, the total number of comparisons done by Algorithm 1. For the number of swaps, Table 1 gives S, wherein I included the swap in line 9 of Algorithm 1, so $S = S_1 + S_2$. The second swap is located after the outer loop, so we simply have $S_2 = A$. Finally, Section 7.3.2 shows that $R = 2A + 1$.

With these cosmetic simplifications, we find the expected total costs in the MMIX cost model for sorting a random permutation by Listing 1 to be

$$T_n^{\mathtt{MMIX}} = A(43\upsilon + 12\mu) + C(4\upsilon + \mu) + S_1(9\upsilon + 2\mu) + 5\upsilon \ . \tag{7.1}$$

7.1.3 MMIX Implementation for Algorithm 8

Listing 2 shows an MMIX implementation of Algorithm 8. Its sheer length may seem scary at first sight, but notice that almost half of the code deals with preparing the arguments and stack management for the recursive calls.

The swap of the pivots in line 3 of Algorithm 8 does not involve array accesses in my MMIX implementation. Instead, I only exchange the contents of the two registers p and q (line 7). Later, Algorithm 8 moves the pivots to their final positions in lines 23 and 24 by two more swaps. In Listing 2, we simply omit reading the pivots from the array and use the register contents instead.

The basic blocks and their cost contribution are shown in Table 7. Determining the frequencies of the mispredicted branches is a bit of a nuisance. However, be assured that all frequencies can be determined by directly applying Kirchhoff's laws to derive the edge flows from the block frequencies. Drawing the relevant parts of the control flow graph on a scratch paper helps a lot.

Almost all block frequencies can be described in terms of swap and comparison location frequencies. The single exception is the block used to evaluate the second part of the conjunction that forms the loop condition in line 11 of Algorithm 8. This block is executed whenever the first part of the condition evaluates to *true*, so its frequency is a new quantity X. It will be discussed in detail in Section 7.3.3.

Applying Knuthian analysis and directly using $R = 3A + 1$ and $S_0 = \frac{1}{2}A$ (see Sections 7.3.1 and 7.3.2) yields the total expected cost of Listing 2 on a random permutation of length n:

$$\begin{aligned} T_n^{\mathtt{MMIX}} = A(\tfrac{129}{2}\upsilon + 18\mu) + C_1(9\upsilon + \mu) + C_3(5\upsilon + \mu) \\ + S_1(5\upsilon + 3\mu) + S_2(7\upsilon + 2\mu) + S_3(6\upsilon + 3\mu) + X \cdot 2\upsilon + 5\upsilon \ . \end{aligned}$$

Due to the asymmetry of Algorithm 8, different swap and comparison locations contribute quite different costs.

Listing 2. MMIX implementation of dual pivot Quicksort with Yaroslavksiy's partitioning method (Algorithm 8).

```
Qsort ⌜CMP   tmp,lleft,rright   R        if left ≥ right terminate.
      ⌞PBNN  tmp,9F             R

      ⌜LDO   p,A,lleft          A        pivot p := A[left]
       LDO   q,A,rright         A        pivot q := A[right]
       CMP   tmp,p,q            A        if p > q
      ⌞BNP   tmp,1F             A        | (Skip if p ≤ q.)

      ⌜SET   tmp,p              S0       | Swap registers p and q
       SET   p,q                S0       | Real swap not needed!
      ⌞SET   q,tmp              S0       |

1H    ⌜ADDU  ll,lleft,8         A        ℓ := left + 1
       SUBU  gg,rright,8        A        g := right − 1
      ⌞SET   kk,ll              A        k := ℓ

loop  ⌜CMP   tmp,kk,gg          C1 + A   while k ≤ g
      ⌞BP    tmp,after          C1 + A   | (Break if k > g)

      ⌜LDO   Ak,A,kk            C1       |
       CMP   tmp,Ak,p           C1       | if A[k] < p
      ⌞PBNN  tmp,2F             C1       | | (Skip if A[k] ≥ p.)

      ⌜LDO   tmp,A,ll           S1       | | Swap A[ℓ] and A[k].
       STO   Ak,A,ll            S1       | |
       STO   tmp,A,kk           S1       | |
       ADDU  ll,ll,8            S1       | | ℓ := ℓ + 1
      ⌞JMP   3F                 S1       | | Skip else-branch.

2H    ⌜CMP   tmp,Ak,q           C1 − S1  | else if A[k] ≥ q
      ⌞PBN   tmp,3F             C1 − S1  | | (Skip if A[k] < q).

4H    ⌜LDO   Ag,A,gg            C3       | | while A[g] > q ∧ k < g
       CMP   tmp,Ag,q           C3       | | | (Break if A[g] ≤ q ...
      ⌞BNP   tmp,5F             C3       | | |

      ⌜CMP   tmp,kk,gg          X        | | | ...or if k ≥ g).
      ⌞PBNN  tmp,5F             X        | | |

      ⌜SUBU  gg,gg,8            C3 − S2  | | | g := g − 1
      ⌞JMP   4B                 C3 − S2  | | end while

5H    ⌜STO   Ak,A,gg            S2       | | Swap A[k] and A[g]
       STO   Ag,A,kk            S2       | | (Ak, Ag now swapped!)
       SUBU  gg,gg,8            S2       | | g := g − 1
       CMP   tmp,Ag,p           S2       | | if A[k] < p (Ag == A[k])
      ⌞PBNN  tmp,3F             S2       | | | (Skip if A[k] ≥ p.)

      ⌜LDO   tmp,A,ll           S3       | | | Swap A[k] and A[l].
       STO   tmp,A,kk           S3       | | |
       STO   Ag,A,ll            S3       | | |
      ⌞ADDU  ll,ll,8            S3       | | | ℓ := ℓ + 1
```

```
3H     ⌜ADDU kk,kk,8             C1       | k := k + 1
       ⌞JMP  loop                C1       end while
after  ⌜SUBU ll,ll,8             A        ℓ := ℓ − 1
        ADDU gg,gg,8             A        g := g + 1
        LDO  tmp,A,ll            A        Swap A[left] and A[ℓ].
        STO  tmp,A,lleft         A
        STO  p,A,ll              A
        LDO  tmp,A,gg            A        Swap A[right] and A[g].
        STO  tmp,A,rright        A
        STO  q,A,gg              A
                                        Recursive Calls
        STO  lleft,sp,0          A        Rescue registers on stack.
        STO  rright,sp,1*8       A
        STO  return,sp,2*8       A
        STO  ll,sp,3*8           A
        STO  gg,sp,4*8           A
                                        Quicksort(A, left, ℓ − 1)
        SUBU arg2,ll,8           A
        ADDU sp,sp,5*8           A        Advance stack pointer.
        GETA argRet,@+8          A        Store return address.
       ⌞JMP  Qsort               A
       ⌜SUBU sp,sp,5*8           A        Pop stored registers.
                                        Quicksort(A, ℓ + 1, g − 1)
        LDO  arg1,sp,3*8         A
        ADDU arg1,arg1,8         A
        LDO  arg2,sp,4*8         A
        SUBU arg2,arg2,8         A
        ADDU sp,sp,5*8           A        Advance stack pointer.
        GETA argRet,@+8          A        Store return address.
       ⌞JMP  Qsort               A
       ⌜SUBU sp,sp,5*8           A        Pop stored registers.
                                        Quicksort(A, g + 1, right)
        LDO  arg1,sp,4*8         A
        ADDU arg1,arg1,8         A
        LDO  arg2,sp,1*8         A
        ADDU sp,sp,5*8           A        Advance stack pointer.
        GETA argRet,@+8          A        Store return address.
       ⌞JMP  Qsort               A
       ⌜SUBU sp,sp,5*8           A        Pop stored registers.
       ⌞LDO  return,sp,2*8       A        Restore return address.
9H     ⌜⌞GO  return,return,0     R        Return to caller.
```

Lines	Frequency	instruction costs	Mispredicted Branch	Frequency
1–2	R	2υ	$2 \rightsquigarrow 3$	A
3–6	A	$4\upsilon + 2\mu$	$6 \rightsquigarrow 10$	$A - S_0$
7–9	S_0	3υ		
10–13	A	4υ		
15–15	$S_3 + A$	υ		
17–18	$C_1 + Y$	2υ	$18 \rightsquigarrow 55$	Y
19–21	C_1	$3\upsilon + \mu$	$21 \rightsquigarrow 22$	S_1
22–26	S_1	$5\upsilon + 3\mu$		
27–28	$C_1 - S_1$	2υ	$28 \rightsquigarrow 31$	$S_3 + A - Y$
29–30	$C_1 - (S_3 + A - Y)$	2υ		
31–31	$S_3 + A - Y$	υ		
32–34	C_3	$3\upsilon + \mu$	$34 \rightsquigarrow 35$	S_2
35–39	S_2	$5\upsilon + 3\mu$		
40–41	$C_3 - S_2$	2υ	$41 \rightsquigarrow 46$	S_3
42–43	$C_3 - S_3$	2υ	$43 \rightsquigarrow 55$	$A - Y$
44–45	$C_3 - (A - Y) - S_3$	2υ		
46–54	S_3	$9\upsilon + 6\mu$		
55–66	A	$11\upsilon + 7\mu$		
67–74	A	$8\upsilon + 2\mu$		
75–81	A	$7\upsilon + 2\mu$		
82–83	A	$2\upsilon + \mu$		
84–84	R	3υ		

Table 8: Basic Block Instruction costs for the MMIX implementation (Listing 3) of dual pivot Quicksort with Kciwegdes partitioning (Algorithm 9). The costs are given in terms of υ ("oops") and μ ("mems"). Moreover, for every conditional jump, the frequency of the mispredicted branch is given.

7.1.4 MMIX Implementation for Algorithm 9

After having managed the 81 lines of Listing 2, the 84 lines of the MMIX implementation of Algorithm 9 in Listing 3 on the facing page should not come as a shock. As for Listing 2, almost half of the code deals with stack management for the recursive calls, which results in easily analyzable sequential code.

The similarities with the implementation of Algorithm 8 continue: Again, the swap of the two pivot elements can be done entirely in registers. Only when p and q are written back to their final positions in lines 27 and 28 of Algorithm 9 do we need the actual array write operation.

It is pleasant to see the clumsy control flow description of the inner loop in Algorithm 9 vanish at machine code level: In pseudocode, the "while true" loop gives the feeling of increased complexity compared to the 'cleaner' inner loops of SEDGEWICK's original partitioning method in Algorithm 7. In MMIX however, both variants of the inner loops consist of an unconditional back jump and some conditional exit branch in the body.

The basic blocks of Listing 3 are summarized in Table 8, including their frequencies and corresponding costs. Most frequencies can be expressed in terms of swap and comparison location frequencies, which we determined in Section 5.1. However, the frequency of some basic blocks depends on how often we leave the outer loop through the break inside the first inner loop. In the new quantity Y, we count how often this happens.

As an example, take line 31 of Listing 3, corresponding to line 15 of Algorithm 9. It is executed for every iteration of the outer loop if we leave through the j-loop, which amounts to frequency $S_3 + A$. However, if we leave the outer loop via the i-loop, line 31 is not reached in this last iteration. So, we find frequency $S_3 + A - Y$ in total.

Table 8 also contains the penalties for mispredicted branches. Their frequencies are easily computed using KIRCHHOFF's laws from the given block frequencies. Drawing the relevant parts of the control flow graph on a scratch paper can be handy.

Using $R = 3A + 1$ and $S_0 = \frac{1}{2}A$ and applying KNUTHian analysis yields the total expected cost of Listing 3 on a random permutation of length n:

$$\begin{aligned} T_n^{\texttt{MMIX}} &= A(\tfrac{115}{2}\upsilon + 14\mu) + C_1(9\upsilon + \mu) + C_3(9\upsilon + \mu) \\ &\qquad + S_1(5\upsilon + 3\mu) + S_2(5\upsilon + 3\mu) + S_3(9\upsilon + 6\mu) + Y \cdot 3\upsilon + 5\upsilon\ . \end{aligned}$$

Listing 3. MMIX implementation of dual pivot Quicksort with Kciwegdes partitioning method (Algorithm 9).

```
Qsort ⌜CMP   tmp,lleft,rright   $R$                  if left $\geqslant$ right terminate
      ⌞PBNN  tmp,9F             $R$
      ⌜LDO   p,A,lleft          $A$                  pivot $p := A[left]$
       LDO   q,A,rright         $A$                  pivot $q := A[right]$
       CMP   tmp,p,q            $A$                  if $p > q$
      ⌞BNP   tmp,1F             $A$                  | (Skip if $p \leqslant q$.)
      ⌜SET   tmp,p              $S_0$                | Swap registers p and q
       SET   p,q                $S_0$                | Real swap not needed!
      ⌞SET   q,tmp              $S_0$                |
1H    ⌜SET   ii,lleft           $A$                  $i := left$
       SET   ii1,lleft          $A$                  $i_1 := left$
       SET   jj,rright          $A$                  $j := right$
      ⌞SET   jj1,rright         $A$                  $j_1 := right$
                                                     while true
loop  ⌜ADDU  ii,ii,8            $S_3 + A$            | $i := i + 1$
                                                     | while true
3H    ⌜CMP   tmp,ii,jj          $C_1 + Y$            | | if $i \geqslant j$
      ⌞BNN   tmp,after          $C_1 + Y$            | | | Break outer while.
      ⌜LDO   Ai,A,ii            $C_1$                | |
       CMP   tmp,Ai,p           $C_1$                | | if $A[i] < p$
      ⌞PBNN  tmp,4F             $C_1$                | | | (Skip if $A[i] \geqslant p$.)
      ⌜STO   Ai,A,ii1           $S_1$                | | | "Hole-move swap"
       ADDU  ii1,ii1,8          $S_1$                | | | $i_1 := i_1 + 1$
       LDO   tmp,A,ii1          $S_1$                | | |
       STO   tmp,A,ii           $S_1$                | | |
      ⌞JMP   7F                 $S_1$                | | | Skip else branch.
4H    ⌜CMP   tmp,Ai,q           $C_1 - S_1$          | | else if $A[i] \geqslant q$
      ⌞BNN   tmp,2F             $C_1 - S_1$          | | | Break inner loop.
7H    ⌜ADDU  ii,ii,8            $C_1 - (S_3 + A - Y)$  | | $i := i + 1$
      ⌞JMP   3B                 $C_1 - (S_3 + A - Y)$  | end while
2H    ⌜SUBU  jj,jj,8            $S_3 + A - Y$        | $j := j - 1$
3H    ⌜LDO   Aj,A,jj            $C_3$                | while true
       CMP   tmp,Aj,q           $C_3$                | | if $A[j] > q$
      ⌞PBNP  tmp,4F             $C_3$                | | | (Skip if $A[j] \leqslant q$.)
      ⌜STO   Aj,A,jj1           $S_2$                | | | "Hole-move swap"
       SUBU  jj1,jj1,8          $S_2$                | | | $j_1 := j_1 - 1$
       LDO   tmp,A,jj1          $S_2$                | | |
       STO   tmp,A,jj           $S_2$                | | |
      ⌞JMP   7F                 $S_2$                | | | Skip else branch.
4H    ⌜CMP   tmp,Aj,p           $C_3 - S_2$          | | else if $A[j] \leqslant p$
```

```
        ⌞BNP   tmp,5F              C3 − S2              | | | Break inner loop.
7H      ⌜CMP   tmp,ii,jj           C3 − S3              | | if i ⩾ j
        ⌞BNN   tmp,after           C3 − S3              | | | Break outer while.
        ⌜SUBU  jj,jj,8             C3 − (A − Y) − S3    | | j := j − 1
        ⌞JMP   3B                  C3 − (A − Y) − S3    | end while
5H      ⌜STO   Aj,A,ii1            S3                   | "Double hole move swap"
         STO   Ai,A,jj1            S3                   |
         ADDU  ii1,ii1,8           S3                   | i1 := i1 + 1
         SUBU  jj1,jj1,8           S3                   | j1 := j1 − 1
         LDO   tmp,A,ii1           S3                   |
         STO   tmp,A,ii            S3                   |
         LDO   tmp,A,jj1           S3                   |
         STO   tmp,A,jj            S3                   |
        ⌞JMP   loop                S3                   end while
after   ⌜STO   p,A,ii1             A                    A[i1] := p
         STO   q,A,jj1             A                    A[j1] := q
                                                       Recursive Calls
         STO   lleft,sp,0          A                      Rescue registers on stack.
         STO   rright,sp,1*8       A
         STO   return,sp,2*8       A
         STO   ii1,sp,3*8          A
         STO   jj1,sp,4*8          A
         SUBU  arg2,ii1,8          A                    Quicksort(A, left, i1 − 1)
         ADDU  sp,sp,5*8           A                      Advance stack pointer.
         GETA  argRet,@+8          A                      Store return address.
        ⌞JMP   Qsort               A
        ⌜SUBU  sp,sp,5*8           A                      Pop stored registers.
         LDO   arg1,sp,3*8         A                    Quicksort(A, i1 + 1, j1 − 1)
         ADDU  arg1,arg1,8         A
         LDO   arg2,sp,4*8         A
         SUBU  arg2,arg2,8         A
         ADDU  sp,sp,5*8           A                      Advance stack pointer.
         GETA  argRet,@+8          A                      Store return address.
        ⌞JMP   Qsort               A
        ⌜SUBU  sp,sp,5*8           A                      Pop stored registers.
         LDO   arg1,sp,4*8         A                    Quicksort(A, j1 + 1, right)
         ADDU  arg1,arg1,8         A
         LDO   arg2,sp,1*8         A
         ADDU  sp,sp,5*8           A                      Advance stack pointer.
         GETA  argRet,@+8          A                      Store return address.
        ⌞JMP   Qsort               A
        ⌜SUBU  sp,sp,5*8           A                      Pop stored registers.
        ⌞LDO   return,sp,2*8       A                      Restore return address.
9H      ⌜GO    return,return,0     R                      Return to caller.
```

Listing 4. Java implementation of classic Quicksort (Algorithm 1).

```
static void Qsort(int[] A, int left, int right) {
   if (right > left) {
      final int p = A[right]; // the pivot
      int i = left - 1, j = right;
      while (true) {
         do ++i; while (A[i] < p);
         do --j; while (A[j] > p);
         if (i >= j) break;
         final int tmp = A[i]; A[i] = A[j]; A[j] = tmp;
      }
      final int tmp = A[i]; A[i] = A[right]; A[right] = tmp;
      Qsort(A, left, i - 1);
      Qsort(A, i + 1, right);
   }
}
```

7.2 Java Bytecode

> *And it was patently obvious that the internet and Java were a match made in heaven. So that's what we did.*
>
> —J. A. Gosling in "Java Technology: An Early History"

Since the release by Sun Microsystems in 1995, the Java programming language and the associated Java Virtual Machine (JVM) have become one of the major platforms of software industry. Arguably the most striking feature of Java is its platform independence—encapsulated in the slogan "Write Once, Run Everywhere". This might have helped the Java technology to co-emerge with the world wide web, where developing for heterogeneous systems has become the default. The choice as main development language for the Android mobile operating system gave Java a further boost.

At the same time, Java has also been widely accepted in academia. Textbooks on algorithms like [SW11] use Java, as do many lectures on programming. Moreover, Java is well-suited for research on programming languages due to its well-defined semantics.

Apart from the success of the Java programming language, the underlying virtual machine has proven to be a stable and quite efficient platform. In more recent years, it has been adopted as target platform for many new programming languages, which shows that Java Bytecode is flexible enough to support different programming paradigms.

As with the Java programming language, the platform's popularity is in part due to the good specification [LY99] of Java Bytecode and the JVM. However, unlike for Knuth's MMIX, no guarantee about the execution time of Bytecode instructions is given. In fact, the portable nature of Java Bytecode renders such global guarantees impossible. Even

Listing 5. Bytecode implementation of classic Quicksort (Algorithm 1). It was obtained by compiling the Java implementation Listing 4 and disassembling the result.

```
Qsort ([III)V :
     ⌜ILOAD          right    R
      ILOAD          left     R
     ⌞IF_ICMPLE      L5       R
     ⌜ALOAD          A        A
      ILOAD          right    A
      IALOAD                  A
      ISTORE         p        A
      ILOAD          left     A
      ICONST_1                A
      ISUB                    A
      ISTORE         i        A
      ILOAD          right    A
     ⌞ISTORE         j        A
L1   ⌜IINC           i,1      $C_1$
      ALOAD          A        $C_1$
      ILOAD          i        $C_1$
      IALOAD                  $C_1$
      ILOAD          p        $C_1$
     ⌞IF_ICMPLT      L1       $C_1$
L2   ⌜IINC           j, -1    $C_2$
      ALOAD          A        $C_2$
      ILOAD          j        $C_2$
      IALOAD                  $C_2$
      ILOAD          p        $C_2$
     ⌞IF_ICMPGT      L2       $C_2$
     ⌜ILOAD          i        $S_1 + A$
      ILOAD          j        $S_1 + A$
     ⌞IF_ICMPLT      L3       $S_1 + A$
     ⌜⌞GOTO          L4       A
L3   ⌜ALOAD          A        $S_1$
      ILOAD          i        $S_1$
      IALOAD                  $S_1$
      ISTORE         tmp      $S_1$
      ALOAD          A        $S_1$
      ILOAD          i        $S_1$
      ALOAD          A        $S_1$
      ILOAD          j        $S_1$
      IALOAD                  $S_1$
      IASTORE                 $S_1$
      ALOAD          A        $S_1$
      ILOAD          j        $S_1$
      ILOAD          tmp      $S_1$
      IASTORE                 $S_1$
     ⌞GOTO           L1       $S_1$
L4   ⌜ALOAD          A        A
      ILOAD          i        A
      IALOAD                  A
      ISTORE         tmp      A
      ALOAD          A        A
      ILOAD          i        A
      ALOAD          A        A
      ILOAD          right    A
      IALOAD                  A
      IASTORE                 A
      ALOAD          A        A
      ILOAD          right    A
      ILOAD          tmp      A
      IASTORE                 A
      ALOAD          A        A
      ILOAD          left     A
      ILOAD          i        A
      ICONST_1                A
      ISUB                    A
     ⌞INVOKESTATIC   Qsort    A
     ⌜ALOAD          A        A
      ILOAD          i        A
      ICONST_1                A
      IADD                    A
      ILOAD          right    A
     ⌞INVOKESTATIC   Qsort    A
L5   ⌜⌞RETURN                 R
```

Lines	Frequency	number of instructions
2–4	R	3
5–14	A	10
15–20	C_1	6
21–26	C_2	6
27–29	$S_1 + A$	3
30–30	A	1
31–45	S_1	15
46–65	A	20
66–71	A	6
72–72	R	1

Table 9: Basic Block Instruction costs for the Java Bytecode implementation (Listing 5) of classic Quicksort (Algorithm 1). The costs of a block are the number of Bytecode instructions in it.

Lines	Frequency	number of instructions
2–6	R	5
7–13	A	7
14–27	S_0	14
28–45	A	18
46–48	$C_1 + A$	3
49–53	C_1	5
54–69	S_1	16
70–74	$C_1 - S_1$	5
75–79	C_3	5
80–82	X	3
83–84	$C_3 - S_2$	2
85–104	S_2	20
105–119	S_3	15
120–121	C_1	2
122–157	A	36
158–165	A	8
166–171	A	6
172–172	R	1

Table 10: Basic Block Instruction costs for the Java Bytecode implementation (Listing 7) of dual pivot Quicksort with YAROSLAVSKIY's partitioning (Algorithm 8). The costs of a block are the number of Bytecode instructions in it.

Listing 6. Java implementation of dual pivot Quicksort with Yaroslavskiy's partitioning method (Algorithm 8).

```
static void Qsort(int[] A, int left, int right) {
    if (right - left >= 1) {
        if (A[left] > A[right]) {
            final int tmp = A[left]; A[left] = A[right]; A[right] = tmp;
        }
        final int p = A[left]; final int q = A[right];
        int l = left + 1, g = right - 1, k = l;
        while (k <= g) {
            if (A[k] < p) {
                final int tmp = A[k]; A[k] = A[l]; A[l] = tmp;
                ++l;
            } else if (A[k] >= q) {
                while (A[g] > q && k < g)  --g;
                {final int tmp = A[k]; A[k] = A[g]; A[g] = tmp;}
                --g;
                if (A[k] < p) {
                    final int tmp = A[k]; A[k] = A[l]; A[l] = tmp;
                    ++l;
                }
            }
            ++k;
        }
        --l; ++g;
        {final int tmp = A[left]; A[left] = A[l]; A[l] = tmp;}
        {final int tmp = A[right]; A[right] = A[g]; A[g] = tmp;}
        Qsort(A, left, l - 1);
        Qsort(A, l + 1, g - 1);
        Qsort(A, g + 1, right);
    }
}
```

for a given machine, runtime predictions are tough — especially since the introduction of just-in-time compilers into the JVM.

However in [CHB06], Camesi et al. experimentally study the correlation between running time and *number of executed Bytecode instructions*. For that, they run a set of benchmark applications and measure runtime and number of executed Bytecodes over time.

They report a quite reliable correlation for a *fixed JVM implementation and a fixed application*, even if the just-in-time compiler of Oracle's JVM implementation is used. This means, that the expected number of executed Bytecodes approximates the runtime of a Java implementation of an algorithm *up to a constant factor*. In particular, the *relative* runtime contributions of different dominant elementary operations can be derived from that. Yet, more empirical evidence is needed, especially with pure algorithms instead of whole software suites to support the hypothesis.

A trivial extension of the cost model might improve the accuracy: Instead of simply counting the *number* of Bytecodes, we can assign a *weight* to each Bytecode instruction, approximating its runtime cost. I do not know a sensible source of such weights and hence confine myself to noting that my analysis trivially generalizes to weighted Bytecode counts.

7.2.1 Remarks for the Java Bytecode Implementations

A full introduction of Java Bytecode is well beyond the scope of this thesis. However, the very readable JVM specification [LY99] can be warmly recommended. It is also available online and has an index of all Bytecode instructions. For the reader familiar with assembly languages it might be enough to wrap up the rough concepts. Java Bytecode is a stack-oriented language. Instructions pop their operands from the stack and push results back onto the stack. In addition, some instructions can access a *local variable* of the current procedure.

The Bytecode programs used here were obtained from the Java source code shown in Listings 4, 6 and 8 using Oracle's Java compiler (javac version 1.7.0_03).

7.2.2 Bytecode Implementation for Algorithm 1

Listing 4 on page 119 shows the code for my Java implementation of classic Quicksort. In comparison with Algorithm 1, I included a tiny optimization: Instead of checking "$j > i$" twice — once before the swap and then in the loop condition — Listing 4 only does this check once before the swap. If it fails, the loop is quit.

The Java compiler produces Listing 5 on page 120 out of Listing 4. As for the MMIX programs, basic blocks are embraced by ⌜⌞ and each instruction is accompanied by its frequency of execution. This information is also summarized in Table 9 on page 121.

Summing over the product of block costs and frequencies gives the total expected costs for an execution of Listing 5:

$$T_n^{\mathrm{JVM}} = 4 \cdot R + 40 \cdot A + 6(C_1 + C_2) + 18 \cdot S_1 \ .$$

Using $R = 2A + 1$ from Section 7.3.2, $C = C_1 + C_2$ and $S_1 = S - A$ for S reported in Table 1 on page 30, we get the final result

$$T_n^{\mathrm{JVM}} = 48 \cdot A + 6 \cdot C + 18 \cdot S_1 + 4 \,.$$

7.2.3 Bytecode Implementation for Algorithm 8

Compiling the straight-forward Java implementation Listing 6 on page 122 of Algorithm 8 yields the slightly lengthy Bytecode program shown in Listing 7. Much of this length is due to the *six* locations $S_0, \dots, S_5$ in Listing 7, where swaps are done: To avoid method invocation overhead, the Bytecode instructions comprising a single swap are copied six times.

Listing 7 induces the basic blocks shown in Table 10 on page 121. The corresponding frequencies and cost contributions are also given. Summing these up and directly using $R = 3A + 1$ (see Section 7.3.2) as well as $S_0 = \frac{1}{2}A$ yields

$$T_n^{\mathrm{JVM}} = 103 \cdot A + 15 \cdot C_1 + 7 \cdot C_3 + 11 \cdot S_1 + 18 \cdot S_2 + 15 \cdot S_3 + 3 \cdot X + 6 \,.$$

It is remarkable that the different swap and comparison locations contribute quite different amounts to the total costs. This is a consequence of the asymmetric nature of Algorithm 8.

Listing 7. Bytecode implementation of dual pivot Quicksort with YAROSLAVSKIY's partitioning method (Algorithm 8). It was obtained by compiling the Java implementation Listing 6 and disassembling the result.

```
QSort ([III)V :
     ⌜ILOAD        right     R
      ILOAD        left      R
      ISUB                   R
      ICONST_1               R
     ⌞IF_ICMPLT    L12       R
     ⌜ALOAD        A         A
      ILOAD        left      A
      IALOAD                 A
      ALOAD        A         A
      ILOAD        right     A
      IALOAD                 A
     ⌞IF_ICMPLE    L1        A
     ⌜ALOAD        A         S0
      ILOAD        left      S0
      IALOAD                 S0
      ISTORE       tmp       S0
      ALOAD        A         S0
      ILOAD        left      S0
      ALOAD        A         S0
      ILOAD        right     S0
      IALOAD                 S0
      IASTORE                S0
      ALOAD        A         S0
      ILOAD        right     S0
      ILOAD        tmp       S0
     ⌞IASTORE                S0
L1   ⌜ALOAD        A         A
      ILOAD        left      A
      IALOAD                 A
      ISTORE       p         A
      ALOAD        A         A
      ILOAD        right     A
      IALOAD                 A
      ISTORE       q         A
      ILOAD        left      A
```

```
        ICONST_1                 A
        IADD                     A
        ISTORE       l           A
        ILOAD        right       A
        ICONST_1                 A
        ISUB                     A
        ISTORE       g           A
        ILOAD        l           A
       ⌞ISTORE       k           A
L2     ⌜ILOAD        k           C1 + A
        ILOAD        g           C1 + A
       ⌞IF_ICMPGT    L11         C1 + A
       ⌜ALOAD        A           C1
        ILOAD        k           C1
        IALOAD                   C1
        ILOAD        p           C1
       ⌞IF_ICMPGE    L3          C1
       ⌜ALOAD        A           S1
        ILOAD        k           S1
        IALOAD                   S1
        ISTORE       tmp         S1
        ALOAD        A           S1
        ILOAD        k           S1
        ALOAD        A           S1
        ILOAD        l           S1
        IALOAD                   S1
        IASTORE                  S1
        ALOAD        A           S1
        ILOAD        l           S1
        ILOAD        tmp         S1
        IASTORE                  S1
        IINC         l,1         S1
       ⌞GOTO         L10         S1
L3     ⌜ALOAD        A           C1 − S1
        ILOAD        k           C1 − S1
        IALOAD                   C1 − S1
        ILOAD        q           C1 − S1
       ⌞IF_ICMPLT    L10         C1 − S1
L4     ⌜ALOAD        A           C3
        ILOAD        g           C3
        IALOAD                   C3
        ILOAD        q           C3
       ⌞IF_ICMPLE    L9          C3
       ⌜ILOAD        k           X
        ILOAD        g           X
       ⌞IF_ICMPGE    L9          X
       ⌜IINC         g,-1        C3 − S2
       ⌞GOTO         L4          C3 − S2
L9     ⌜ALOAD        A           S2
        ILOAD        k           S2
        IALOAD                   S2
        ISTORE       tmp         S2
        ALOAD        A           S2
        ILOAD        k           S2
        ALOAD        A           S2
        ILOAD        g           S2
        IALOAD                   S2
        IASTORE                  S2
        ALOAD        A           S2
        ILOAD        g           S2
        ILOAD        tmp         S2
        IASTORE                  S2
        IINC         g,-1        S2
        ALOAD        A           S2
        ILOAD        k           S2
        IALOAD                   S2
        ILOAD        p           S2
       ⌞IF_ICMPGE    L10         S2
       ⌜ALOAD        A           S3
        ILOAD        k           S3
        IALOAD                   S3
        ISTORE       tmp         S3
        ALOAD        A           S3
        ILOAD        k           S3
        ALOAD        A           S3
        ILOAD        l           S3
        IALOAD                   S3
        IASTORE                  S3
        ALOAD        A           S3
        ILOAD        l           S3
        ILOAD        tmp         S3
        IASTORE                  S3
       ⌞IINC         l,1         S3
L10    ⌜IINC         k,1         C1
       ⌞GOTO         L2          C1
L11    ⌜IINC         l,-1        A
        IINC         g,1         A
        ALOAD        A           A
        ILOAD        left        A
        IALOAD                   A
        ISTORE       tmp         A
        ALOAD        A           A
        ILOAD        left        A
        ALOAD          A        A
        ILOAD          l        A
        IALOAD                  A
        IASTORE                 A
        ALOAD          A        A
        ILOAD          l        A
        ILOAD          tmp      A
        IASTORE                 A
        ALOAD          A        A
        ILOAD          right    A
        IALOAD                  A
        ISTORE         tmp      A
        ALOAD          A        A
        ILOAD          right    A
        ALOAD          A        A
        ILOAD          g        A
        IALOAD                  A
        IASTORE                 A
        ALOAD          A        A
        ILOAD          g        A
        ILOAD          tmp      A
        IASTORE                 A
        ALOAD          A        A
        ILOAD          left     A
        ILOAD          l        A
        ICONST_1                A
        ISUB                    A
       └INVOKESTATIC   Qsort    A
       ┌ALOAD          A        A
        ILOAD          l        A
        ICONST_1                A
        IADD                    A
        ILOAD          g        A
        ICONST_1                A
        ISUB                    A
       └INVOKESTATIC   Qsort    A
       ┌ALOAD          A        A
        ILOAD          g        A
        ICONST_1                A
        IADD                    A
        ILOAD          right    A
       └INVOKESTATIC   Qsort    A
L12    ⌈RETURN                  R
```

7.2.4 Bytecode Implementation for Algorithm 9

Finally, I also wrote a Java implementation of Algorithm 9. It is shown in Listing 8 on the facing page. Compiling it to Bytecode yields Listing 9 on page 129. Again, I inlined the swap instructions, such that many of the instructions in Listing 9 are found at the swap locations $S_0, \ldots, S_5$.

Listing 9 induces the basic blocks shown in Table 11 on page 128. As usual, we sum over all these blocks and take the product of frequency and cost contribution, directly incorporating $R = 3A + 1$ and $S_0 = \frac{1}{2}A$. The total costs of Listing 9 are

$$T_n^{\mathrm{JVM}} = 70 \cdot A + 15(C_1 + C_3) + 9 \cdot (S_1 + S_2) + 24 \cdot S_3 + 5 \cdot Y + 4\,.$$

Listing 8. Java implementation of dual pivot Quicksort with Kciwegdes partitioning (Algorithm 9).

```
static void Qsort(int[] A, int left, int right) {
    if (right - left >= 1) {
        if (A[left] > A[right]) {
            final int tmp = A[left]; A[left] = A[right]; A[right] = tmp;
        }
        final int p = A[left]; final int q = A[right];
        int i = left, i1 = left, j = right, j1 = right;
outer:  while (true) {
            ++i;
            while (true) {
                if (i >= j) break outer;
                if (A[i] < p) {
                    A[i1] = A[i]; ++i1; A[i] = A[i1];
                } else if (A[i] >= q) break;
                ++i;
            }
            --j;
            while (true) {
                if (A[j] > q) {
                    A[j1] = A[j]; --j1; A[j] = A[j1];
                } else if (A[j] <= p) break;
                if (i >= j) break outer;
                --j;
            }
            A[i1] = A[j]; A[j1] = A[i];
            ++i1; --j1;
            A[i] = A[i1]; A[j] = A[j1];
        }
        A[i1] = p;
        A[j1] = q;
        Qsort(A, left, l - 1);
        Qsort(A, l + 1, g - 1);
        Qsort(A, g + 1, right);
    }
}
```

Lines	Frequency	number of instructions
2–4	R	3
5–11	A	7
12–25	S_0	14
26–41	A	16
42–42	$S_3 + A$	1
43–45	$C_1 + Y$	3
46–46	Y	1
47–51	C_1	5
52–65	S_1	14
66–70	$C_1 - S_1$	5
71–71	$S_3 + A - Y$	1
72–73	$C_1 - (S_3 + A - Y)$	2
74–74	$S_3 + A - Y$	1
75–79	C_3	5
80–93	S_2	14
94–98	$C_3 - S_2$	5
99–99	S_3	1
100–102	$C_3 - S_3$	3
103–103	$A - Y$	1
104–105	$C_3 - (A - Y) - S_3$	2
106–132	S_3	27
133–146	A	14
147–154	A	8
155–160	A	6
161–161	R	1

Table 11: Basic Block Instruction costs for the Java Bytecode implementation (Listing 9) of dual pivot Quicksort with KCIWEGDES partitioning (Algorithm 9). The costs of a block are the number of Bytecode instructions in it.

Listing 9. Bytecode implementation of dual pivot Quicksort with KCIWEGDES partitioning (Algorithm 9). It was obtained by compiling the Java implementation Listing 8 and disassembling the result.

```
QSort ([III)V :
      ⌜ILOAD        right      R
       ILOAD        left       R
      ⌞IF_ICMPLE    L14        R
      ⌜ALOAD        A          A
       ILOAD        left       A
       IALOAD                  A
       ALOAD        A          A
       ILOAD        right      A
       IALOAD                  A
      ⌞IF_ICMPLE    L1         A
      ⌜ALOAD        A          S_0
       ILOAD        left       S_0
       IALOAD                  S_0
       ISTORE       tmp        S_0
       ALOAD        A          S_0
       ILOAD        left       S_0
       ALOAD        A          S_0
       ILOAD        right      S_0
       IALOAD                  S_0
       IASTORE                 S_0
       ALOAD        A          S_0
       ILOAD        right      S_0
       ILOAD        tmp        S_0
      ⌞IASTORE                 S_0
L1    ⌜ILOAD        left       A
       ISTORE       i          A
       ILOAD        left       A
       ISTORE       i1         A
       ILOAD        right      A
       ISTORE       j          A
       ILOAD        right      A
       ISTORE       j1         A
       ALOAD        A          A
       ILOAD        left       A
       IALOAD                  A
       ISTORE       p          A
       ALOAD        A          A
       ILOAD        right      A
       IALOAD                  A
      ⌞ISTORE       q          A
L2    ⌜⌞IINC        i,1        S_3 + A
L3    ⌜ILOAD        i          C_1 + Y
       ILOAD        j          C_1 + Y
      ⌞IF_ICMPLT    L4         C_1 + Y
      ⌜⌞GOTO        L13        Y
L4    ⌜ALOAD        A          C_1
       ILOAD        i          C_1
       IALOAD                  C_1
       ILOAD        p          C_1
      ⌞IF_ICMPGE    L5         C_1
      ⌜ALOAD        A          S_1
       ILOAD        i1         S_1
       ALOAD        A          S_1
       ILOAD        i          S_1
       IALOAD                  S_1
       IASTORE                 S_1
       IINC         i1 1       S_1
       ALOAD        A          S_1
       ILOAD        i          S_1
       ALOAD        A          S_1
       ILOAD        i1         S_1
       IALOAD                  S_1
       IASTORE                 S_1
      ⌞GOTO         L6         S_1
L5    ⌜ALOAD        A          C_1 - S_1
       ILOAD        i          C_1 - S_1
       IALOAD                  C_1 - S_1
       ILOAD        q          C_1 - S_1
      ⌞IF_ICMPLT    L6         C_1 - S_1
      ⌜⌞GOTO        L7         S_3 + A - Y
L6    ⌜IINC         i 1        C_1 - (S_3 + A - Y)
      ⌞GOTO         L3         C_1 - (S_3 + A - Y)
L7    ⌜⌞IINC        j,-1       S_3 + A - Y
L8    ⌜ALOAD        A          C_3
       ILOAD        j          C_3
       IALOAD                  C_3
       ILOAD        q          C_3
      ⌞IF_ICMPLE    L9         C_3
      ⌜ALOAD        A          S_2
```

```
        ILOAD          j1        S2
        ALOAD          A         S2
        ILOAD          j         S2
        IALOAD                   S2
        IASTORE                  S2
        IINC           j1,-1     S2
        ALOAD          A         S2
        ILOAD          j         S2
        ALOAD          A         S2
        ILOAD          j1        S2
        IALOAD                   S2
        IASTORE                  S2
        GOTO           L10       S2
L9      ALOAD          A         C3 − S2
        ILOAD          j         C3 − S2
        IALOAD                   C3 − S2
        ILOAD          p         C3 − S2
        IF_ICMPGT      L10       C3 − S2
        GOTO           L12       S3
L10     ILOAD          i         C3 − S3
        ILOAD          j         C3 − S3
        IF_ICMPLT      L11       C3 − S3
        GOTO           L13       A − Y
L11     IINC           j,-1      C3 − (A − Y) − S3
        GOTO           L8        C3 − (A − Y) − S3
L12     ALOAD          A         S3
        ILOAD          i1        S3
        ALOAD          A         S3
        ILOAD          j         S3
        IALOAD                   S3
        IASTORE                  S3
        ALOAD          A         S3
        ILOAD          j1        S3
        ALOAD          A         S3
        ILOAD          i         S3
        IALOAD                   S3
        IASTORE                  S3
        IINC           i1,1      S3
        IINC           j1,-1     S3
        ALOAD          A         S3
        ILOAD          i         S3
        ALOAD          A         S3
        ILOAD          i1        S3
        IALOAD                   S3
        IASTORE                  S3
        ALOAD          A         S3
        ILOAD          j         S3
        ALOAD          A         S3
        ILOAD          j1        S3
        IALOAD                   S3
        IASTORE                  S3
        GOTO           L2        S3
L13     ALOAD          A         A
        ILOAD          i1        A
        ILOAD          p         A
        IASTORE                  A
        ALOAD          A         A
        ILOAD          j1        A
        ILOAD          q         A
        IASTORE                  A
        ALOAD          A         A
        ILOAD          left      A
        ILOAD          i1        A
        ICONST_1                 A
        ISUB                     A
        INVOKESTATIC   Qsort     A
        ALOAD          A         A
        ILOAD          i1        A
        ICONST_1                 A
        IADD                     A
        ILOAD          j1        A
        ICONST_1                 A
        ISUB                     A
        INVOKESTATIC   Qsort     A
        ALOAD          A         A
        ILOAD          j1        A
        ICONST_1                 A
        IADD                     A
        ILOAD          right     A
        INVOKESTATIC   Qsort     A
L14     RETURN                   R
```

7.3 Analyzing The Missing Frequencies

Thanks to the farsighted decision in Chapter 4 to compute the expected frequencies *separately per swap and comparison location*, most of the quantities occurring in the basic block frequencies above are already known. Tables 3 and 5 summarizes the results and Table 1 gives the results for classic Quicksort. Nevertheless, a few quantities remain to be determined.

7.3.1 Number of Partitions

7.3.1.1 Classic Quicksort

The number of partitioning steps is analyzed e. g. in [Sed77b]. The result is given on page 334, where it says (notation adapted)

$$A = 2\frac{n+1}{M+2} - 1\,,$$

where a specialized sorting method is used lists of size $\leqslant M$ (cf. Section 3.4.3). For our Algorithm 1, we have $M = 1$, so

$$A = \tfrac{2}{3}n - \tfrac{1}{3}\,.$$

7.3.1.2 Dual Pivot Quicksort

If we set $a = 0$, $b = 1$ and $d = 1$ in eq. (4.3) on page 60, the partitioning costs are $pc_n = [n \geqslant 2]$. This is exactly the behavior for the "number of partitioning steps per partitioning step": For primitively sorted lists, we omit the partitioning altogether. For the parameters above, eq. (4.4) on page 61 yields

$$\begin{aligned} A &= \tfrac{2}{5}(n+1) - \tfrac{1}{2}\,. \\ &= \tfrac{2}{5}n - \tfrac{1}{10}\,. \end{aligned}$$

For both Algorithms 8 and 9 we can express some more frequencies in terms of A, the number of partitioning steps: $C_0 = S_4 = S_5 = A$ and $S_0 = \frac{1}{2}A$.

7.3.2 Number of Recursive Calls

All considered Quicksort algorithms skip the partitioning step for lists of length $\leqslant 1$ — such lists are trivially sorted, anyway. Yet, we invoke the corresponding recursive call, which causes some overhead. In the implementations considered here, some basic blocks are executed for every recursive call, even if the partitioning step is omitted. Their frequency is R.

There is a simple argument to express R in terms of A, the number of 'real' partitioning steps: For classic one-pivot Quicksort, every partitioning step causes two additional recursive calls. Moreover, there is one additional call — namely the initial call which started the whole sorting process. Hence, $R = 2A + 1$. For dual pivot Quicksort, each partitioning step invokes three recursive calls, so we get $R = 3A + 1$.

7.3.3 X in Algorithm 8

In the implementations of Algorithm 8, we get one basic block whose frequency is not directly expressible in terms of swap and comparison markers. It results from the loop condition in line 11 of Algorithm 8, which is the conjunction of $\mathrm{A}[g] > q$ and $k < g$. I decided to implement the check as *non-strict* conjunction: If the first operand evaluates to *false*, the second is not evaluated at all.

In Java, the `&&`-operator has exactly these semantics, so the loop condition becomes `A[g] > q && k < g` there (line 13 of Listing 6). The Java compiler translated that to the Bytecode instructions starting at line 75 of Listing 7. I used the same scheme in the MMIX implementation, as well (line 25 of Listing 2).

In both cases, there is a basic block with unknown frequency X, which evaluates the second part of the loop condition — but only if the first part was *true*. The loop body is executed $\mathtt{C}_3 - \mathtt{S}_2$ times, so $X = \mathtt{C}_3 - \mathtt{S}_2 + X'$, where X' is the number of times the loop is left because of $k \not< g$, i. e. $k \geqslant g$.

The expected costs for the whole sorting process depend *linearly* on the expected costs for the *first partitioning step*, see eq. (4.2). Hence, it suffices to determine the expected contribution to X' in the first partitioning step. Let us call this contribution x', i. e. x' is the expected number of times we leave the inner loop because of $k \geqslant g$ in the first partitioning step.

Leaving the inner loop at line 11 because of the second condition was the key to the proof of Lemma 4.3 on page 71 and indeed, the same arguments will help here, as well. The reader is gently advised to recall Lemma 4.3 and its proof, if the following revision appears sketchy.

During one partitioning step, k only increases and stops with $k = g + 1 + \delta$, where $\delta \in \{0, 1\}$. This holds at the end of the outer loop — for the inner loop at line 11, this means that, at any time, $k \leqslant g$. Accordingly, we can leave this loop at most once because of $k \geqslant g$ (per partitioning step). Moreover, if $\delta = 0$, we always have $k < g$ at line 11.

So, we only get contributions to x' for $\delta = 1$. Now, Lemma 4.3 says that $\delta = 1$ if and only if $\mathrm{A}[q] > q$. As we have $g = q$ for the last execution of the inner loop, we *must* leave it via $k \geqslant g$, as $\mathrm{A}[g] = \mathrm{Q}[q] > q$. Consequently, we have $x' = 1$ in this case. Incorporating the case $\delta = 0$, we finally find

$$x' = \delta .$$

Using eq. (4.4), we find $X' = \frac{1}{10}n - \frac{1}{15}$ and finally

$$\begin{aligned} X &= \mathtt{C}_3 - \mathtt{S}_2 + X' \\ &= \tfrac{1}{5}(n+1)\mathcal{H}_{n+1} - \tfrac{39}{100}(n+1) + \tfrac{1}{6} . \end{aligned}$$

7.3.4 Y in Algorithm 9

The implementations of Algorithm 9 induce a new quantity Y, which counts how often we leave the outer loop through the break inside the *first* inner loop. More precisely, Y is the expected number of partitioning steps, where the condition in line 9 of Algorithm 9 is eventually true.

Using the definitions of Section 5.1.1 on page 95, the contribution of the first partitioning step to the overall value of Y is exactly Φ, whose expected value is given in eq. (5.1): $\mathbb{E}\,\Phi = \frac{1}{2} + \frac{1}{n(n-1)}$ for $n \geqslant 2$. Using $pc_n := [n \geqslant 2] \cdot \mathbb{E}\,\Phi$ in the closed form eq. (4.2) on page 59 of the recurrence relation, gives the expectation of Y

$$Y = \tfrac{4}{15}n + \tfrac{1}{60}\ .$$

7.4 Results & Discussion

The exact expected costs for the MMIX and Java Bytecode implementations are given in Table 12. The overall results are quite clear and the same for both MMIX and JVM implementations: Classic Quicksort uses by far the least number of instructions for sorting large random permutations. Among the two dual pivot Quicksort variants, Yaroslavskiy's partitioning is slightly better, at least for large n. However, the two are rather close.

Let us make this more quantitative. First, I consider the number of executed Bytecodes. Here in fact, Classic Quicksort is strictly better than both dual pivot variants for $n \geqslant 13$. Among the two dual pivot Quicksorts, Yaroslavskiy's partitioning is more efficient for large n: For $n \leqslant n_0 := 1535$, Listing 9 causes less Bytecode instructions, for $n > n_0$, Listing 7 is better.

The MMIX costs cannot be compared directly, as they are two-dimensional. However, under the plausible assumption that $\mu = \alpha \cdot \upsilon$ for some constant $\alpha \geqslant 0$, we can rank the costs. For reasonable values of $\alpha \in [0, 100]$, we find Algorithm 9 is surprisingly fastest for very small n. The turnover occurs between 29 and 35 depending on α. Then, for medium sized lists, Algorithm 8 is fastest. The turnover point n^* grows very fast with α. For $n > n^*$, Algorithm 1 is again the fastest. Here are some numerically computed values for n^*

α	0	1	1.5	2	2.5	3	4	5	10	20
n^*	225	501	719	1009	1388	1877	3272	5400	36729	339 086

The bottom line is that under the detailed cost model of counting primitive instructions, the savings in terms of comparisons — which both dual pivot Quicksort variants achieve — is outweighed by the many additional swaps they incur. Of course, this picture might totally change, if the sorting keys are no longer plain integers. For example, if every key comparison has to compare two strings lexicographically, it might be worthwhile to save some of these comparisons at the cost of more swaps.
In order to estimate the impact of such a change, we can try to squeeze out some information about how the two elementary operations of Quicksort — swaps and comparisons — contribute to the overall cost of an execution. The next section pursues this goal and reveals some of the differences of the implementations in passing.

Implementation	T_n with symbolic frequencies
MMIX Classic Quicksort	$(43\upsilon + 12\mu)A + (4\upsilon + \mu)\mathtt{C} + (9\upsilon + 2\mu)\mathtt{S}_1 + 5\upsilon$
MMIX Dual Pivot YAROSLAVSKIY	$(\frac{129}{2}\upsilon + 18\mu)A + (9\upsilon + \mu)\mathtt{C}_1 + (5\upsilon + \mu)\mathtt{C}_3$ $+ (5\upsilon + 3\mu)\mathtt{S}_1 + (7\upsilon + 2\mu)\mathtt{S}_2 + (6\upsilon + 3\mu)\mathtt{S}_3 + 2\upsilon \cdot X + 5\upsilon$
MMIX Dual Pivot KCIWEGDES	$(\frac{115}{2}\upsilon + 14\mu)A + (9\upsilon + \mu)(\mathtt{C}_1 + \mathtt{C}_3)$ $+ (5\upsilon + 3\mu)(\mathtt{S}_1 + \mathtt{S}_2) + (9\upsilon + 6\mu)\mathtt{S}_3 + 3\upsilon \cdot Y + 5\upsilon$
Bytecode Classic Quicksort	$48 \cdot A + 6 \cdot \mathtt{C} + 18 \cdot \mathtt{S}_1 + 4$
Bytecode Dual Pivot YAROSLAVSKIY	$103 \cdot A + 15 \cdot \mathtt{C}_1 + 7 \cdot \mathtt{C}_3 + 11 \cdot \mathtt{S}_1 + 18 \cdot \mathtt{S}_2 + 15 \cdot \mathtt{S}_3 + 3 \cdot X + 6$
Bytecode Dual Pivot KCIWEGDES	$70 \cdot A + 15(\mathtt{C}_1 + \mathtt{C}_3) + 9 \cdot (\mathtt{S}_1 + \mathtt{S}_2) + 24 \cdot \mathtt{S}_3 + 5 \cdot Y + 4$

Implementation	T_n with inserted frequencies (valid for $n \geqslant 4$)
MMIX Classic Quicksort	$(11\upsilon + 2.\overline{6}\mu)(n+1)\mathcal{H}_n + (11\upsilon + 3.\overline{7}\mu)n + (-11.5\upsilon - 4.\overline{5}\mu)$
MMIX Dual Pivot YAROSLAVSKIY	$(13.1\upsilon + 2.8\mu)(n+1)\mathcal{H}_n + (-1.695\upsilon + 1.24\mu)n$ $+ (-1.678\overline{3}\upsilon - 1.79\overline{3}\mu)$
MMIX Dual Pivot KCIWEGDES	$(14.\overline{6}\upsilon + 3.6\mu)(n+1)\mathcal{H}_n + (-7.\overline{2}\upsilon + -2.18\overline{6}\mu)n$ $+ (-1.0\overline{5}\upsilon + -1.33\overline{6}\mu)$
Bytecode Classic Quicksort	$18(n+1)\mathcal{H}_n + 2n - 15$
Bytecode Dual Pivot YAROSLAVSKIY	$23.8(n+1)\mathcal{H}_n - 8.71n - 4.74\overline{3}$
Bytecode Dual Pivot KCIWEGDES	$26(n+1)\mathcal{H}_n - 26.13n - 3.13$

Table 12: Total expected costs of the Quicksort implementations for MMIX and JVM. The upper table lists the cost where the block frequencies are still given symbolically. In the lower table, I inserted the expected values for the frequencies from Tables 1 and 3 and Section 7.3. For better comparability, I use (repeating) decimal representations. I would like to stress that all results are *exact*, not rounded.

7.4.1 Distributing Costs to Elementary Operations

The goal of this section is to 'distribute' the total costs obtained for the various implementations to the different elementary operations considered in Chapter 4. This might help to understand differences between algorithms and identify potential bottlenecks.

7.4.1.1 Relative Runtime Contributions in Algorithm 1

For the MMIX implementation of classic Quicksort, each comparisons contributes $\mathrm{cost}(\mathtt{C}) = 4\upsilon + \mu$ to the total costs on average, whereas a swap costs $\mathrm{cost}(\mathtt{S}) = 9\upsilon + 2\mu$. Assuming a fixed ratio between the cost units, i. e. $\mu = \alpha \cdot \upsilon$ for some $\alpha \geqslant 0$, we get for the relative cost

$$\frac{\mathrm{cost}(\mathtt{S})}{\mathrm{cost}(\mathtt{C})} = \frac{9+2\alpha}{4+\alpha} \in [2, 2.25] \qquad \text{for any } \alpha \geqslant 0 \,.$$

So, for this implementation, swaps are a little bit more expensive than two comparisons. Recall from Table 1 on page 30 that classic Quicksort asymptotically does 6 times as many comparisons as swaps. This indicates that we can make classic Quicksort significantly better if we can save some comparisons.

For the Java Bytecode implementation, we have $\mathrm{cost}(\mathtt{C}) = 6$ and $\mathrm{cost}(\mathtt{S}) = 18$, hence

$$\frac{\mathrm{cost}(\mathtt{S})}{\mathrm{cost}(\mathtt{C})} = 3 \,.$$

Here, swaps are, relatively speaking, slightly more expensive than in MMIX, but still the main bottleneck is formed by comparisons.

7.4.1.2 Relative Runtime Contributions in Algorithm 8

The asymmetry of Algorithm 8 leads to different cost contributions for different swap and comparison locations. As a consequence, we can only compare *expected cost contributions* of swaps and comparisons. The different locations are not reached equally often, and the relative proportions depend on n. For the sake of simplicity, I confine myself to the limiting case $n \to \infty$, i. e. I weight a location's contribution by the *asymptotic* proportion of the frequency of this location among the total number of operations. More formally, I compute $\lim_{n\to\infty} \frac{C_i}{C}$ respectively $\lim_{n\to\infty} \frac{S_i}{S}$ from the frequencies in Table 3:

C_0	C_1	C_2	C_3	C_4
0	$8/19$	$5/19$	$4/19$	$2/19$

S_0	S_1	S_2	S_3	S_4	S_5
0	$1/2$	$1/3$	$1/6$	0	0

Now we need the cost contributions of all locations with weight > 0. For MMIX, we find

$$\mathrm{cost}(\mathtt{C}_1) = 7\upsilon + \mu, \qquad \mathrm{cost}(\mathtt{C}_2) = 2\upsilon, \qquad \mathrm{cost}(\mathtt{C}_3) = 5\upsilon + \mu, \qquad \mathrm{cost}(\mathtt{C}_4) = 7\upsilon + 2\mu$$
$$\mathrm{cost}(\mathtt{S}_1) = 7\upsilon + 3\mu, \qquad \mathrm{cost}(\mathtt{S}_2) = 7\upsilon + 2\mu, \qquad \mathrm{cost}(\mathtt{S}_3) = 6\upsilon + 3\mu \,.$$

All terms for the swaps and the last one for comparisons contain the branch misprediction penalty of 2υ. Similarly, Listing 7 gives the following Bytecode contributions

$$\begin{gathered}\mathrm{cost}(\mathtt{C}_1) = 10, \qquad \mathrm{cost}(\mathtt{C}_2) = 5, \qquad \mathrm{cost}(\mathtt{C}_3) = 7, \qquad \mathrm{cost}(\mathtt{C}_4) = 20 \\ \mathrm{cost}(\mathtt{S}_1) = 16, \qquad \mathrm{cost}(\mathtt{S}_2) = 20, \qquad \mathrm{cost}(\mathtt{S}_3) = 15\ .\end{gathered}$$

For classic Quicksort, we had the fortunate situation that every basic block contained at most either a comparison or a swap. Therefore, we had no trouble distributing blocks to either swaps or comparisons. In Algorithm 8, however, swaps and comparisons appear together: The swap in line 12 and the comparison in line 14 fall into the same basic block. This implies that $\mathtt{C}_4 = \mathtt{S}_2$ — as already noticed in Section 4.3.3.9 — and hence $\mathrm{cost}(\mathtt{C}_4) = \mathrm{cost}(\mathtt{S}_2)$.

How to deal with that? Should we simply attribute the cost contribution to both swaps and comparisons? Rather not, since we would count the corresponding block twice, then. In fact, there seems to be no ultimately satisfying solution for this. The cleanest solution I could find is to introduce a new chimera elementary operation: The dreaded **swapcomp**, which first swaps two elements and then does a comparison on one of them. Hence, the block where both a comparison and a swap happens, is *neither* assigned to the swaps *nor* to the comparisons, but rather to this new kind of elementary operation. I write $\mathtt{SC} = \mathtt{C}_4 = \mathtt{S}_2$ for the frequency of the corresponding block.

For the MMIX implementation, we then find the following average contributions for swaps, comparisons and swapcomps:

$$\begin{aligned}\mathrm{cost}(\mathtt{C}) &= \lim_{n\to\infty} \frac{\mathtt{C}_1\mathrm{cost}(\mathtt{C}_1) + \mathtt{C}_2\mathrm{cost}(\mathtt{C}_2) + \mathtt{C}_3\mathrm{cost}(\mathtt{C}_3)}{\mathtt{C}_1 + \mathtt{C}_2 + \mathtt{C}_3} \\ &= \tfrac{8}{17}(7\upsilon + \mu) + \tfrac{5}{17}\cdot 2\upsilon + \tfrac{4}{17}(5\upsilon + \mu) \\ &= \tfrac{86}{17}\upsilon + \tfrac{12}{17}\mu \quad\approx\quad 5.06\upsilon + 0.71\mu \\ \mathrm{cost}(\mathtt{S}) &= \lim_{n\to\infty} \frac{\mathtt{S}_1\mathrm{cost}(\mathtt{S}_1) + \mathtt{S}_3\mathrm{cost}(\mathtt{S}_3)}{\mathtt{S}_1 + \mathtt{S}_3} \\ &= \tfrac{3}{4}(7\upsilon + 3\mu) + \tfrac{1}{4}(6\upsilon + 3\mu) \\ &= \tfrac{27}{4}\upsilon + 3\mu \quad\approx\quad 6.75\upsilon + 3\mu \\ \mathrm{cost}(\mathtt{SC}) &= 7\upsilon + 2\mu\ .\end{aligned}$$

Similarly, one computes the contributions in the JVM cost model

$$\begin{aligned}\mathrm{cost}(\mathtt{C}) &= \tfrac{8}{17}\cdot 10 + \tfrac{5}{17}\cdot 5 + \tfrac{4}{17}\cdot 7 \\ &= \tfrac{133}{17} \quad\approx\quad 7.82 \\ \mathrm{cost}(\mathtt{S}) &= \tfrac{3}{4}\cdot 16 + \tfrac{1}{4}\cdot 15 \\ &= \tfrac{63}{4} \quad\approx\quad 15.75 \\ \mathrm{cost}(\mathtt{SC}) &= 20\ .\end{aligned}$$

The picture is remarkably less clear than for classic Quicksort. Swaps are still distinctly more expensive. For the MMIX implementation, the difference in υs is rather small, but swaps need much more μs. Note in particular that we use less than one memory access per comparison in expectation, which is due to clever caching of array elements in registers. By exploiting the same trick, the swapcomp costs roughly as much as a half comparison plus a half swap.

The Bytecode implementation cannot use registers to store array accesses. Therefore, a swap typically has to load all elements from memory.[21] Hence, a single swap already amounts to 14 Bytecodes. The remaining ones are overhead like pointer increments and control flow instructions. Thus, it comes at no surprise, that the ratio of swap costs vs. comparison costs is higher than for MMIX, unless MMIX runtime is dominated by memory accesses.

7.4.1.3 Relative Runtime Contributions in Algorithm 9

Even though Algorithm 9 is much more symmetric than Algorithm 8, there still are 'inner' and 'outer' locations, which contribute different amounts of costs. As above, I will weight the different locations by their asymptotic relative frequencies $\lim_{n\to\infty} \frac{C_i}{C}$ respectively $\lim_{n\to\infty} \frac{S_i}{S}$.

C_0	C_1	C_2	C_3	C_4
0	$9/28$	$5/28$	$9/28$	$5/28$

S_0	S_1	S_2	S_3	S_4	S_5
0	$2/5$	$2/5$	$1/5$	0	0

Note that S_3 corresponds to *two* swaps. Since we are looking for the expected cost contribution of a *single* randomly selected swap, its relative weight in the average has to be doubled and its cost halved.

For the MMIX cost contributions from Listing 3, we find

$$\mathrm{cost}(C_1) = \mathrm{cost}(C_3) = 9\upsilon + \mu, \qquad \mathrm{cost}(C_2) = \mathrm{cost}(C_4) = 2\upsilon$$
$$\mathrm{cost}(S_1) = \mathrm{cost}(S_2) = 7\upsilon + 3\mu, \qquad \mathrm{cost}(S_3) = 13\upsilon + 6\mu\,.$$

The terms for the swaps contain branch misprediction penalties of 2υ. Luckily, each basic block contains either a comparison or a swap — or neither of them, so we can unambiguously assign cost contributions. Together, we obtain

$$\begin{aligned}
\mathrm{cost}(C) &= \lim_{n\to\infty} \frac{C_1\mathrm{cost}(C_1) + C_2\mathrm{cost}(C_2) + C_3\mathrm{cost}(C_3) + C_4\mathrm{cost}(C_4)}{C_1 + C_2 + C_3 + C_4} \\
&= (\tfrac{9}{28} + \tfrac{9}{28})(9\upsilon + \mu) + (\tfrac{5}{28} + \tfrac{5}{28})(2\upsilon) \\
&= \tfrac{73}{14}\upsilon + \tfrac{9}{14}\mu \quad \approx \quad 5.21\upsilon + 0.64\mu \\
\mathrm{cost}(S) &= \lim_{n\to\infty} \frac{S_1\mathrm{cost}(S_1) + S_2\mathrm{cost}(S_2) + 2S_3\frac{\mathrm{cost}(S_3)}{2}}{S_1 + S_2 + 2S_3} \\
&= (\tfrac{1}{3} + \tfrac{1}{3})(7\upsilon + 3\mu) + \tfrac{1}{3}\frac{13\upsilon + 6\mu}{2} \\
&= \tfrac{41}{6}\upsilon + 3\mu \quad \approx \quad 6.83\upsilon + 3\mu\,.
\end{aligned}$$

[21] We might use local variables to cache array elements. However, in Bytecode, we would also need to push the contents of local variables on the stack before we can work with them. So, even if both array elements already reside in local variables, 8 Bytecode instructions are needed for the swap.

Similarly, Listing 9 gives the following Bytecode contributions

$$\begin{gathered}\mathrm{cost}(\mathtt{C}_1) = \mathrm{cost}(\mathtt{C}_3) = 10, \qquad \mathrm{cost}(\mathtt{C}_2) = \mathrm{cost}(\mathtt{C}_4) = 5\\ \mathrm{cost}(\mathtt{S}_1) = \mathrm{cost}(\mathtt{S}_2) = 14, \qquad \mathrm{cost}(\mathtt{S}_3) = 27\ .\end{gathered}$$

From those, we again compute the average cost contributions

$$\begin{aligned}\mathrm{cost}(\mathtt{C}) &= \lim_{n\to\infty} \frac{\mathtt{C}_1\mathrm{cost}(\mathtt{C}_1) + \mathtt{C}_2\mathrm{cost}(\mathtt{C}_2) + \mathtt{C}_3\mathrm{cost}(\mathtt{C}_3) + \mathtt{C}_4\mathrm{cost}(\mathtt{C}_4)}{\mathtt{C}_1 + \mathtt{C}_2 + \mathtt{C}_3 + \mathtt{C}_4}\\ &= (\tfrac{9}{28} + \tfrac{9}{28})(10) + (\tfrac{5}{28} + \tfrac{5}{28})(5)\\ &= \tfrac{115}{14} \quad\approx\quad 8.21\\ \mathrm{cost}(\mathtt{S}) &= \lim_{n\to\infty} \frac{\mathtt{S}_1\mathrm{cost}(\mathtt{S}_1) + \mathtt{S}_2\mathrm{cost}(\mathtt{S}_2) + 2\mathtt{S}_3\frac{\mathrm{cost}(\mathtt{S}_3)}{2}}{\mathtt{S}_1 + \mathtt{S}_2 + 2\mathtt{S}_3}\\ &= (\tfrac{1}{3} + \tfrac{1}{3})(14) + \tfrac{1}{3}\frac{27}{2}\\ &= \tfrac{83}{3} \quad\approx\quad 13.83\ .\end{aligned}$$

As for Algorithm 8, the MMIX implementation behaves differently in the two cost dimensions: The difference in υs is rather small, whereas swaps need much more μs. As explained in the last section, this is partly due to caching of array elements.

For the JVM implementation, one swap costs as much as $\frac{581}{345} \approx 1.68$ comparisons. This ratio is much smaller than for both Algorithms 1 and 8. In fact, the "hole-move" type swaps used in Algorithm 9 do not require temporary storage, which allows a swap to be done in 12 Bytecodes. This low ratio explains, why Listing 9 remains rather competitive despite the many extra swaps it needs.

8 Runtime Study

> “ *A man with a watch knows what time it is. A man with two watches is never sure.* ”
>
> — Segal’s Law

Chapters 4 and 5 showed that the dual pivot Quicksort variants save key comparisons, but need more swaps. In Chapter 7, we determined the expected number of primitive instructions performed by implementations of the algorithms on two machines — with the result that on those machines, swaps are so expensive that they outweigh the savings in comparisons.

This rises the question, whether dual pivot Quicksort is competitive on real machines, if we consider *actual running time*. After all, Yaroslavskiy’s dual pivot Quicksort was chosen as standard sorting method for Oracle’s Java 7 runtime library based on such runtime tests!

Those tests were based on an optimized implementation intended for production use. For example, the original code in the runtime library selects the *tertiles of five elements* as pivots. Such additional variations might have distorted the results of those runtime studies. To eliminate such secondary influences, this chapter presents a runtime study directly based on Algorithms 1, 8 and 9.

The particular optimization of pivot sampling is investigated in Chapter 9. There, we will show that in terms of the number of primitive instructions, selecting the tertiles of five elements is not as helpful for Algorithm 8 as one might think at first sight. In fact, a slight variation of the pivot sampling scheme improves the expected number of instructions by more than 3 % without any additional costs. This shows that one can very easily fall for premature optimization when applying variations of Quicksort. This is another important reason for studying the runtime of the basic algorithms in isolation.

8.1 Setup for the Experiments

Here I describe as briefly as possible, yet hopefully sufficiently completely, my setup for the runtime study.

8.1.1 Machine & Software

I used straight-forward implementations of Algorithms 1, 7, 8 and 9 in Java and C++. The Java programs were compiled using javac version 1.7.0_03 from the Oracle Java Development Kit and run on the HotSpot 64-bit Server VM, version 1.7.0_03. For C++, I used the GNU C++ compiler g++ version 4.4.5 with optimization level 3 (`-O3`).

To keep the effort reasonable, I confine myself to one test machine, even though influences of different processors, operating systems etc. would be interesting to investigate. The computer has an Intel Core i7 920 processor with four cores with hyperthreading, running at 2.67GHz. This processor has 8MB of shared on-die L3 cache. The system has 6GB of main memory. The operating system is Ubuntu 10.10 with Linux 2.6.35-32-generic kernel. Whilst running the simulations, graphical user interface was completely disabled to have as little background services running as possible.

8.1.2 Input Generation

I created a test bed of random permutations once and run all algorithms on these same inputs. This "common random number" method was suggested in [McG92] as a variance reduction technique. The lists were created as follows: For every input size $n \in \mathit{sizes}$, with

$$\begin{aligned}\mathit{sizes} := \{10^1, 10^2, 10^3, 10^4, 10^5, 2.5 \cdot 10^5, 5 \cdot 10^5, 7.5 \cdot 10^5,\\ 10^6, 1.25 \cdot 10^6, 1.5 \cdot 10^6, 1.75 \cdot 10^6, 2 \cdot 10^6\},\end{aligned}$$

I created 1000 random permutations by the following folklore algorithm, shown here as Java code.

```
int[] A = new int[len];
for (int i = 1; i <= len; ++i)
    A[i - 1] = i;
for (int i = len - 1; i > 1; --i)
    swap(A, i - 1, random.nextInt(i));
```

If `random.nextInt(i)` provides perfectly random uniformly distributed integers in $[0..i-1]$, the above code creates every permutation of $[1..\texttt{len}]$ with the same probability. As real random numbers are not available or at least too expensive, I used the Mersenne twister pseudo random number generator proposed in [MN98]. The used Java implementation is due to Luke and available from `http://cs.gmu.edu/~sean/research/`.

8.1.3 Runtime Measurement Methodology

Measuring running times of programs is a nuisance. In principle, one only needs to record time before and after a run, then the difference is the running time. However, the devil is in the detail. The first question is the notion of time. We have *wall clock time* — the actual time period passed according to the clock attached to you office wall — versus *processor time*, the time that the processor actually spent executing *your* program. I measure processor time to alleviate disturbances by other programs.

The next task is to find processor timers that offer sufficient resolution and accuracy. Sorting a single list for the considered sizes takes between a few and a few hundred milliseconds. For C++, I used the Linux runtime library, specifically the function `clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &res);` from `sys/time.h`. On the test machine, its claims to have nanosecond accuracy, which sounds very optimistic. It reported reasonable time spans in a small test.

For Java, I could not find a suitable processor timer with high accuracy. However, the function `ManagementFactory.getThreadMXBean().getThreadCpuTime(threadId)` can give the processor time of a given thread with 10 ms accuracy. So, I repeat every single sorting process until it takes some seconds to run in total in a new thread. Then, the above function is called for this new thread. For time spans of a few seconds, 10 ms accuracy provide measurements with ±1 % error, which is acceptable. The total time is then divided by the number of repetitions. This increases the overall time for simulations, but provides reliable measurements.

I conducted some experiments to assess the quality of both methods of runtime measurements. To this end, I ran the same algorithm on the same input many times and recorded every single running time. In a perfect environment, every such run should take exactly the same time and the variance should be zero. The actual result is then somewhat disillusioning: A spread of about ±2 % was observed repeatedly. Moreover, a typical measurement contains a few outliers that took much more time than the rest. Presumably those outliers are due to interferences of task switches and interrupts by other processes.

This liability of runtime measurements to noise should be taken into account when interpreting experimental results. For example, the actual variance in runtimes due to different lists of the same size is most probably buried by measurement noise. However, the sample mean provides a usable estimate of the actual expected runtime.

8.2 Runtime Comparison

Using the setup described in the last section, I conducted my runtime study of all considered sorting methods on three different *runtime platforms*[22]:

- **Java**
  The Oracle HotSpot 64-bit Server VM with just-in-time Bytecode compiler.
- **C++**
  C++ implementation compiled to native binary using `g++ -O3`.
- **Java -Xint**
  The Oracle HotSpot 64-bit Server VM in pure interpretive mode, i. e. with just-in-time Bytecode compiler *disabled*. This is achieved by `java -Xint`.
  Interpretive mode of modern JVMs is typically one order of magnitude slower than just-in-time compiled code. Therefore, it is no longer used in production systems. I nevertheless included it here, as this runtime platform is the closest we can get to the runtime model of counting executed Bytecode instructions. As interpretive mode is much slower, only 100 instead of 1000 inputs per size were used.

In addition to implementations of Algorithms 1, 7, 8 and 9, I also report running times for the sorting method of the corresponding programming library. For Java, this is a

[22] "Runtime platform" might not be the best umbrella term for these three settings. Improvements are welcome.

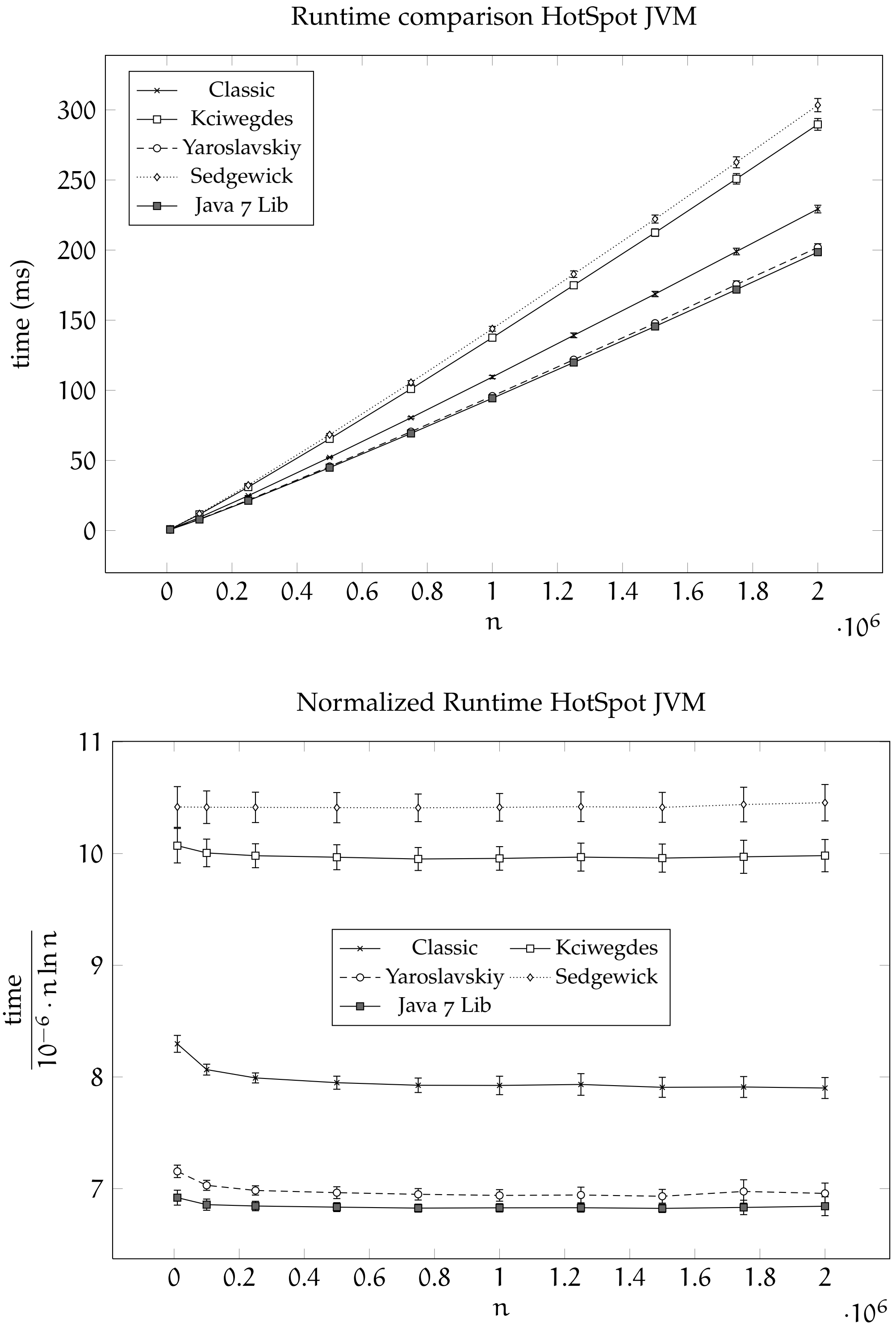


Figure 5: Comparison of the actual runtime for the Java implementation of our Quicksort variants. Also included is the sorting method Oracle ships in the Java 7 runtime library, abbreviated as "Java 7 Lib". The upper plot shows absolute runtimes, the lower plot the same data, but normalized through division by $n \ln n$. Both plot contain error bars that indicate one standard deviation.

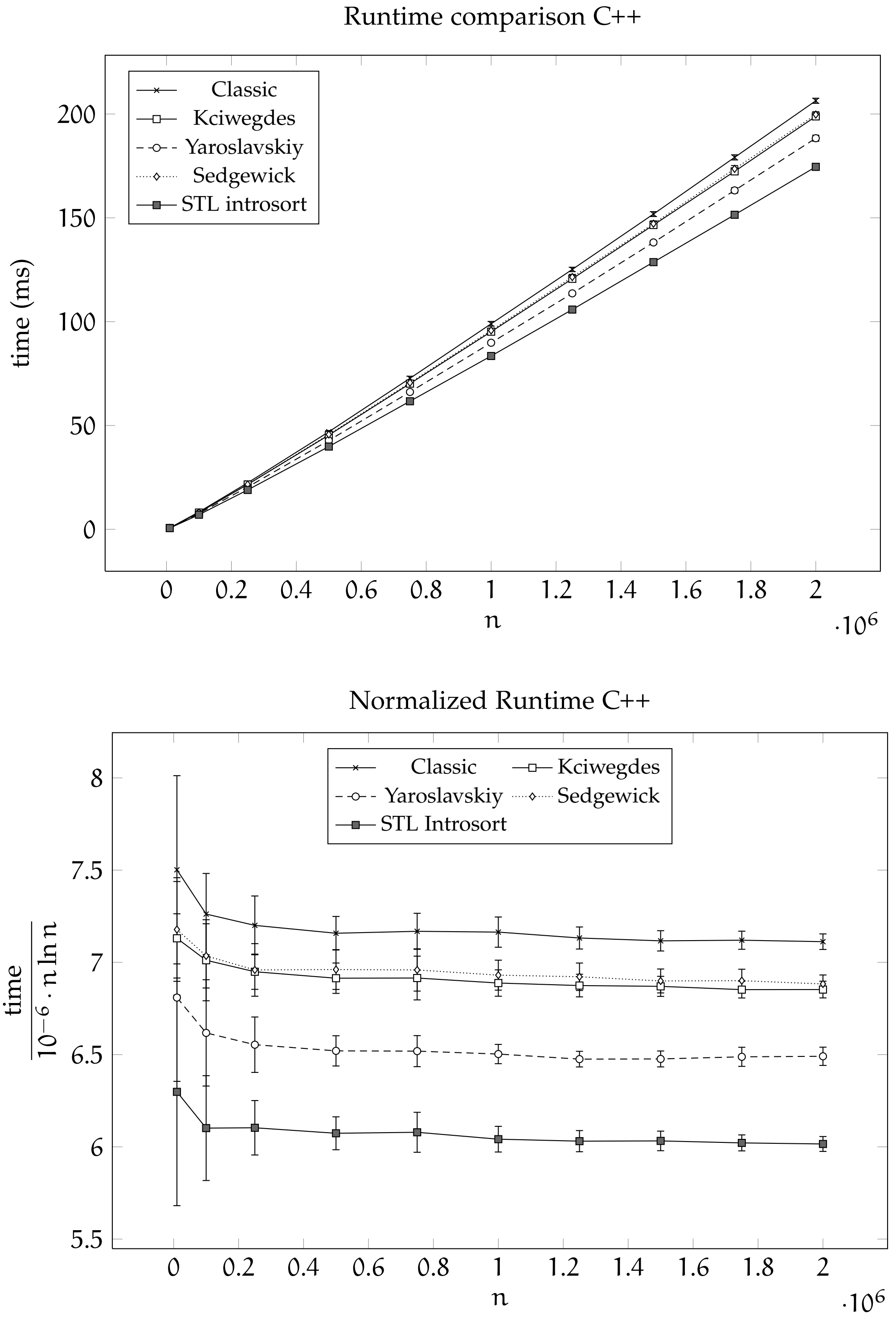
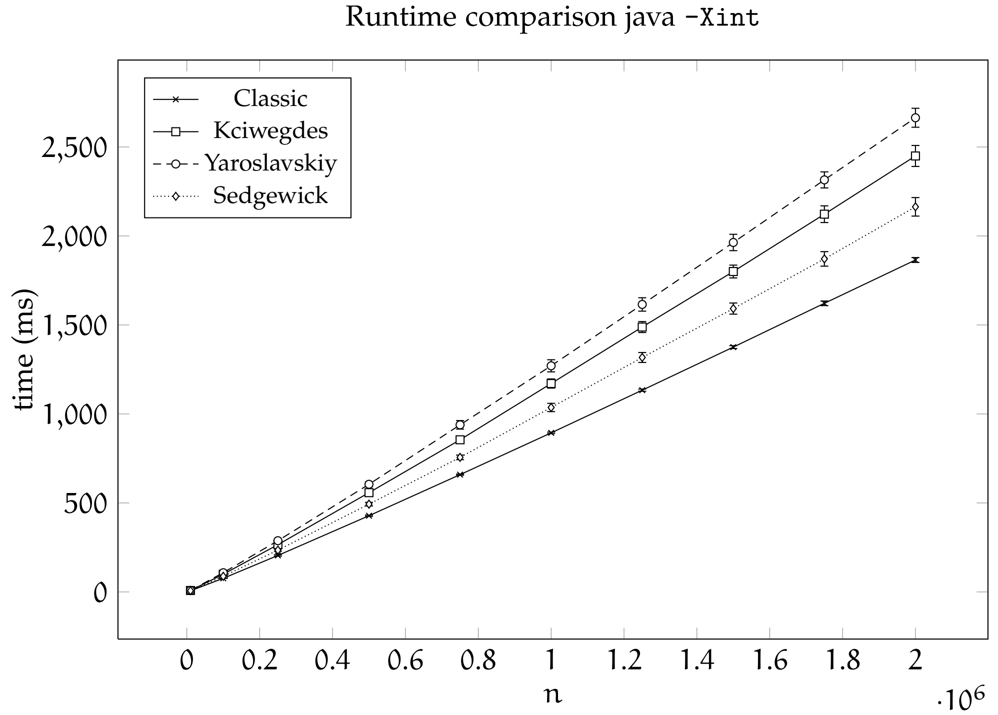


Figure 6: Comparison of the actual runtime for the C++ implementation of our Quicksort variants. Also included is "STL introsort", the sorting method from the algorithms part of the Standard Template Library. The upper plot shows absolute runtimes, the lower plot the same data, but normalized through division by $n \ln n$. Both plot contain error bars that indicate one standard deviation.



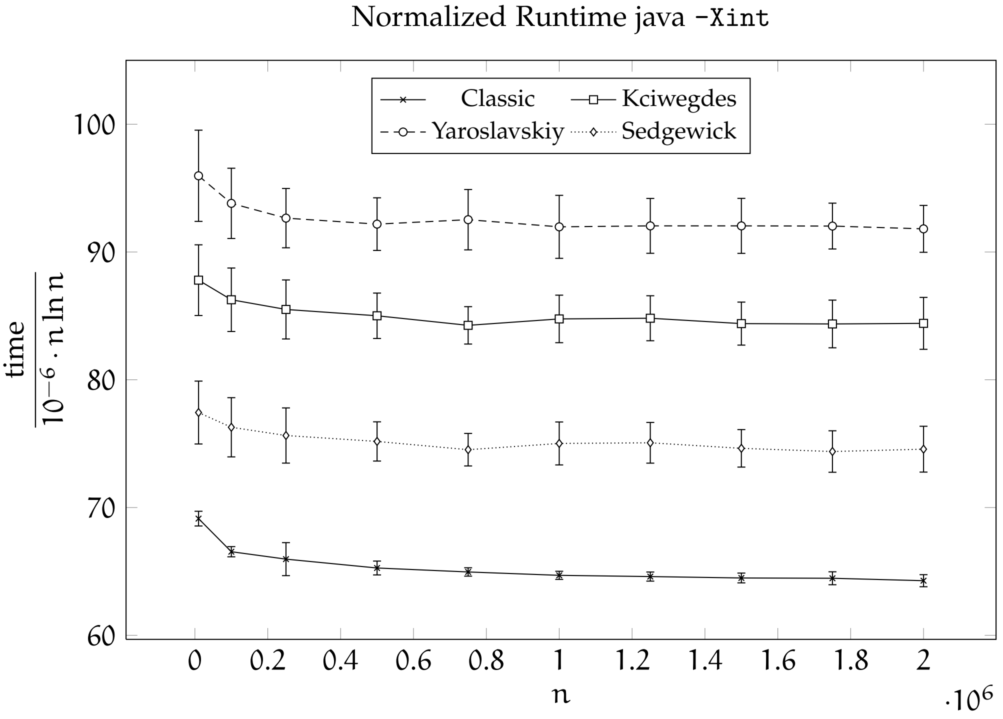


Figure 7: Comparison of the actual runtime for the Java implementation of our Quicksort variants, run in *interpretive mode*. The upper plot shows absolute runtimes, the lower plot the same data, but normalized through division by $n \ln n$. Both plot contain error bars that indicate one standard deviation. For interpretive mode, only 100 inputs per size were used.

n	Classic	Kciwegdes	Yaroslavskiy	Sedgewick	Java 7 Lib
$1{\cdot}10^1$	$1.09 \cdot 10^{-4}$ $(9.4 \cdot 10^{-6})$	$1.35 \cdot 10^{-4}$ $(1.3 \cdot 10^{-5})$	$1.08 \cdot 10^{-4}$ $(1.1 \cdot 10^{-5})$	$1.25 \cdot 10^{-4}$ $(1.2 \cdot 10^{-5})$	$7.84 \cdot 10^{-5}$ $(1.1 \cdot 10^{-5})$
$1{\cdot}10^2$	$2.14 \cdot 10^{-3}$ $(1.9 \cdot 10^{-4})$	$2.48 \cdot 10^{-3}$ $(1.5 \cdot 10^{-4})$	$1.95 \cdot 10^{-3}$ $(2.0 \cdot 10^{-4})$	$2.78 \cdot 10^{-3}$ $(2.4 \cdot 10^{-4})$	$1.92 \cdot 10^{-3}$ $(3.0 \cdot 10^{-4})$
$1{\cdot}10^3$	$5.73 \cdot 10^{-2}$ $(9.3 \cdot 10^{-4})$	$6.68 \cdot 10^{-2}$ $(1.5 \cdot 10^{-3})$	$4.76 \cdot 10^{-2}$ $(6.4 \cdot 10^{-4})$	$6.76 \cdot 10^{-2}$ $(1.7 \cdot 10^{-3})$	$4.52 \cdot 10^{-2}$ $(7.6 \cdot 10^{-4})$
$1{\cdot}10^4$	$7.64 \cdot 10^{-1}$ $(7.0 \cdot 10^{-3})$	$9.28 \cdot 10^{-1}$ $(1.4 \cdot 10^{-2})$	$6.59 \cdot 10^{-1}$ $(5.1 \cdot 10^{-3})$	$9.59 \cdot 10^{-1}$ $(1.7 \cdot 10^{-2})$	$6.37 \cdot 10^{-1}$ $(6.2 \cdot 10^{-3})$
$1{\cdot}10^5$	$9.29 \cdot 10^{0}$ $(5.6 \cdot 10^{-2})$	$1.15 \cdot 10^{1}$ $(1.4 \cdot 10^{-1})$	$8.09 \cdot 10^{0}$ $(5.2 \cdot 10^{-2})$	$1.20 \cdot 10^{1}$ $(1.7 \cdot 10^{-1})$	$7.89 \cdot 10^{0}$ $(5.8 \cdot 10^{-2})$
$2.5{\cdot}10^5$	$2.48 \cdot 10^{1}$ $(1.4 \cdot 10^{-1})$	$3.10 \cdot 10^{1}$ $(3.3 \cdot 10^{-1})$	$2.17 \cdot 10^{1}$ $(1.3 \cdot 10^{-1})$	$3.24 \cdot 10^{1}$ $(4.2 \cdot 10^{-1})$	$2.13 \cdot 10^{1}$ $(1.3 \cdot 10^{-1})$
$5{\cdot}10^5$	$5.22 \cdot 10^{1}$ $(3.8 \cdot 10^{-1})$	$6.54 \cdot 10^{1}$ $(7.3 \cdot 10^{-1})$	$4.57 \cdot 10^{1}$ $(3.5 \cdot 10^{-1})$	$6.83 \cdot 10^{1}$ $(8.9 \cdot 10^{-1})$	$4.48 \cdot 10^{1}$ $(2.6 \cdot 10^{-1})$
$7.5{\cdot}10^5$	$8.04 \cdot 10^{1}$ $(6.6 \cdot 10^{-1})$	$1.01 \cdot 10^{2}$ $(1.0 \cdot 10^{0})$	$7.05 \cdot 10^{1}$ $(5.2 \cdot 10^{-1})$	$1.06 \cdot 10^{2}$ $(1.3 \cdot 10^{0})$	$6.93 \cdot 10^{1}$ $(3.6 \cdot 10^{-1})$
$1{\cdot}10^6$	$1.09 \cdot 10^{2}$ $(1.1 \cdot 10^{0})$	$1.38 \cdot 10^{2}$ $(1.5 \cdot 10^{0})$	$9.59 \cdot 10^{1}$ $(7.1 \cdot 10^{-1})$	$1.44 \cdot 10^{2}$ $(1.7 \cdot 10^{0})$	$9.43 \cdot 10^{1}$ $(5.1 \cdot 10^{-1})$
$1.25{\cdot}10^6$	$1.39 \cdot 10^{2}$ $(1.7 \cdot 10^{0})$	$1.75 \cdot 10^{2}$ $(2.2 \cdot 10^{0})$	$1.22 \cdot 10^{2}$ $(1.2 \cdot 10^{0})$	$1.83 \cdot 10^{2}$ $(2.3 \cdot 10^{0})$	$1.20 \cdot 10^{2}$ $(6.8 \cdot 10^{-1})$
$1.5{\cdot}10^6$	$1.69 \cdot 10^{2}$ $(1.9 \cdot 10^{0})$	$2.12 \cdot 10^{2}$ $(2.7 \cdot 10^{0})$	$1.48 \cdot 10^{2}$ $(1.3 \cdot 10^{0})$	$2.22 \cdot 10^{2}$ $(2.9 \cdot 10^{0})$	$1.46 \cdot 10^{2}$ $(7.9 \cdot 10^{-1})$
$1.75{\cdot}10^6$	$1.99 \cdot 10^{2}$ $(2.4 \cdot 10^{0})$	$2.51 \cdot 10^{2}$ $(3.7 \cdot 10^{0})$	$1.75 \cdot 10^{2}$ $(2.6 \cdot 10^{0})$	$2.63 \cdot 10^{2}$ $(3.9 \cdot 10^{0})$	$1.72 \cdot 10^{2}$ $(1.6 \cdot 10^{0})$
$2{\cdot}10^6$	$2.29 \cdot 10^{2}$ $(2.7 \cdot 10^{0})$	$2.90 \cdot 10^{2}$ $(4.2 \cdot 10^{0})$	$2.02 \cdot 10^{2}$ $(2.7 \cdot 10^{0})$	$3.03 \cdot 10^{2}$ $(4.7 \cdot 10^{0})$	$1.99 \cdot 10^{2}$ $(2.4 \cdot 10^{0})$

Table 13: Raw runtime data for Java underlying the plots in Figure 5. Every entry gives the mean of the runs on 1000 lists of this size and the corresponding sample standard deviation in parentheses. All times are given in milliseconds.

n	Classic	Kciwegdes	Yaroslavskiy	Sedgewick	STL Introsort
$1{\cdot}10^1$	$5.49 \cdot 10^{-4}$ $(1.6 \cdot 10^{-4})$	$3.66 \cdot 10^{-4}$ $(5.1 \cdot 10^{-5})$	$3.68 \cdot 10^{-4}$ $(5.1 \cdot 10^{-5})$	$3.89 \cdot 10^{-4}$ $(5.4 \cdot 10^{-5})$	$3.83 \cdot 10^{-4}$ $(3.3 \cdot 10^{-4})$
$1{\cdot}10^2$	$4.06 \cdot 10^{-3}$ $(3.5 \cdot 10^{-3})$	$4.19 \cdot 10^{-3}$ $(1.2 \cdot 10^{-3})$	$3.59 \cdot 10^{-3}$ $(5.6 \cdot 10^{-4})$	$3.82 \cdot 10^{-3}$ $(2.4 \cdot 10^{-3})$	$3.74 \cdot 10^{-3}$ $(1.1 \cdot 10^{-3})$
$1{\cdot}10^3$	$5.37 \cdot 10^{-2}$ $(8.0 \cdot 10^{-3})$	$5.23 \cdot 10^{-2}$ $(1.0 \cdot 10^{-2})$	$4.84 \cdot 10^{-2}$ $(3.6 \cdot 10^{-3})$	$5.26 \cdot 10^{-2}$ $(8.3 \cdot 10^{-3})$	$4.40 \cdot 10^{-2}$ $(6.1 \cdot 10^{-3})$
$1{\cdot}10^4$	$6.91 \cdot 10^{-1}$ $(4.7 \cdot 10^{-2})$	$6.57 \cdot 10^{-1}$ $(2.8 \cdot 10^{-2})$	$6.27 \cdot 10^{-1}$ $(4.2 \cdot 10^{-2})$	$6.61 \cdot 10^{-1}$ $(2.6 \cdot 10^{-2})$	$5.80 \cdot 10^{-1}$ $(5.7 \cdot 10^{-2})$
$1{\cdot}10^5$	$8.36 \cdot 10^{0}$ $(2.5 \cdot 10^{-1})$	$8.07 \cdot 10^{0}$ $(2.5 \cdot 10^{-1})$	$7.62 \cdot 10^{0}$ $(3.3 \cdot 10^{-1})$	$8.10 \cdot 10^{0}$ $(2.0 \cdot 10^{-1})$	$7.02 \cdot 10^{0}$ $(3.3 \cdot 10^{-1})$
$2.5{\cdot}10^5$	$2.24 \cdot 10^{1}$ $(5.0 \cdot 10^{-1})$	$2.16 \cdot 10^{1}$ $(3.0 \cdot 10^{-1})$	$2.04 \cdot 10^{1}$ $(4.7 \cdot 10^{-1})$	$2.16 \cdot 10^{1}$ $(4.4 \cdot 10^{-1})$	$1.90 \cdot 10^{1}$ $(4.6 \cdot 10^{-1})$
$5{\cdot}10^5$	$4.70 \cdot 10^{1}$ $(6.0 \cdot 10^{-1})$	$4.54 \cdot 10^{1}$ $(5.4 \cdot 10^{-1})$	$4.28 \cdot 10^{1}$ $(5.4 \cdot 10^{-1})$	$4.57 \cdot 10^{1}$ $(7.1 \cdot 10^{-1})$	$3.98 \cdot 10^{1}$ $(5.9 \cdot 10^{-1})$
$7.5{\cdot}10^5$	$7.27 \cdot 10^{1}$ $(9.9 \cdot 10^{-1})$	$7.02 \cdot 10^{1}$ $(1.2 \cdot 10^{0})$	$6.61 \cdot 10^{1}$ $(8.5 \cdot 10^{-1})$	$7.06 \cdot 10^{1}$ $(1.2 \cdot 10^{0})$	$6.17 \cdot 10^{1}$ $(1.1 \cdot 10^{0})$
$1{\cdot}10^6$	$9.90 \cdot 10^{1}$ $(1.1 \cdot 10^{0})$	$9.52 \cdot 10^{1}$ $(9.9 \cdot 10^{-1})$	$8.98 \cdot 10^{1}$ $(7.2 \cdot 10^{-1})$	$9.58 \cdot 10^{1}$ $(1.1 \cdot 10^{0})$	$8.35 \cdot 10^{1}$ $(9.6 \cdot 10^{-1})$
$1.25{\cdot}10^6$	$1.25 \cdot 10^{2}$ $(1.0 \cdot 10^{0})$	$1.21 \cdot 10^{2}$ $(1.1 \cdot 10^{0})$	$1.14 \cdot 10^{2}$ $(7.4 \cdot 10^{-1})$	$1.21 \cdot 10^{2}$ $(1.3 \cdot 10^{0})$	$1.06 \cdot 10^{2}$ $(1.0 \cdot 10^{0})$
$1.5{\cdot}10^6$	$1.52 \cdot 10^{2}$ $(1.2 \cdot 10^{0})$	$1.47 \cdot 10^{2}$ $(1.2 \cdot 10^{0})$	$1.38 \cdot 10^{2}$ $(9.2 \cdot 10^{-1})$	$1.47 \cdot 10^{2}$ $(1.4 \cdot 10^{0})$	$1.29 \cdot 10^{2}$ $(1.1 \cdot 10^{0})$
$1.75{\cdot}10^6$	$1.79 \cdot 10^{2}$ $(1.2 \cdot 10^{0})$	$1.72 \cdot 10^{2}$ $(1.1 \cdot 10^{0})$	$1.63 \cdot 10^{2}$ $(1.3 \cdot 10^{0})$	$1.74 \cdot 10^{2}$ $(1.6 \cdot 10^{0})$	$1.51 \cdot 10^{2}$ $(1.1 \cdot 10^{0})$
$2{\cdot}10^6$	$2.06 \cdot 10^{2}$ $(1.2 \cdot 10^{0})$	$1.99 \cdot 10^{2}$ $(1.3 \cdot 10^{0})$	$1.88 \cdot 10^{2}$ $(1.4 \cdot 10^{0})$	$2.00 \cdot 10^{2}$ $(1.4 \cdot 10^{0})$	$1.75 \cdot 10^{2}$ $(1.2 \cdot 10^{0})$

Table 14: Raw runtime data for C++ underlying the plots in Figure 6. Every entry gives the mean of the runs on 1000 lists of this size and the corresponding sample standard deviation in parentheses. All times are given in milliseconds.

n	Classic	Kciwegdes	Yaroslavskiy	Sedgewick
$1{\cdot}10^1$	$2.71 \cdot 10^{-3}$ $(2.2 \cdot 10^{-4})$	$2.91 \cdot 10^{-3}$ $(3.4 \cdot 10^{-4})$	$3.42 \cdot 10^{-3}$ $(3.4 \cdot 10^{-4})$	$2.62 \cdot 10^{-3}$ $(2.8 \cdot 10^{-4})$
$1{\cdot}10^2$	$3.82 \cdot 10^{-2}$ $(8.2 \cdot 10^{-4})$	$4.54 \cdot 10^{-2}$ $(2.7 \cdot 10^{-3})$	$5.13 \cdot 10^{-2}$ $(3.1 \cdot 10^{-3})$	$4.00 \cdot 10^{-2}$ $(2.4 \cdot 10^{-3})$
$1{\cdot}10^3$	$5.07 \cdot 10^{-1}$ $(4.9 \cdot 10^{-3})$	$6.31 \cdot 10^{-1}$ $(2.6 \cdot 10^{-2})$	$6.96 \cdot 10^{-1}$ $(3.1 \cdot 10^{-2})$	$5.56 \cdot 10^{-1}$ $(2.8 \cdot 10^{-2})$
$1{\cdot}10^4$	$6.37 \cdot 10^{0}$ $(5.3 \cdot 10^{-2})$	$8.09 \cdot 10^{0}$ $(2.5 \cdot 10^{-1})$	$8.84 \cdot 10^{0}$ $(3.3 \cdot 10^{-1})$	$7.13 \cdot 10^{0}$ $(2.3 \cdot 10^{-1})$
$1{\cdot}10^5$	$7.66 \cdot 10^{1}$ $(4.6 \cdot 10^{-1})$	$9.93 \cdot 10^{1}$ $(2.9 \cdot 10^{0})$	$1.08 \cdot 10^{2}$ $(3.2 \cdot 10^{0})$	$8.78 \cdot 10^{1}$ $(2.7 \cdot 10^{0})$
$2.5{\cdot}10^5$	$2.05 \cdot 10^{2}$ $(4.0 \cdot 10^{0})$	$2.66 \cdot 10^{2}$ $(7.2 \cdot 10^{0})$	$2.88 \cdot 10^{2}$ $(7.2 \cdot 10^{0})$	$2.35 \cdot 10^{2}$ $(6.7 \cdot 10^{0})$
$5{\cdot}10^5$	$4.28 \cdot 10^{2}$ $(3.6 \cdot 10^{0})$	$5.58 \cdot 10^{2}$ $(1.2 \cdot 10^{1})$	$6.05 \cdot 10^{2}$ $(1.4 \cdot 10^{1})$	$4.93 \cdot 10^{2}$ $(1.0 \cdot 10^{1})$
$7.5{\cdot}10^5$	$6.59 \cdot 10^{2}$ $(3.3 \cdot 10^{0})$	$8.55 \cdot 10^{2}$ $(1.5 \cdot 10^{1})$	$9.39 \cdot 10^{2}$ $(2.4 \cdot 10^{1})$	$7.56 \cdot 10^{2}$ $(1.3 \cdot 10^{1})$
$1{\cdot}10^6$	$8.94 \cdot 10^{2}$ $(4.4 \cdot 10^{0})$	$1.17 \cdot 10^{3}$ $(2.6 \cdot 10^{1})$	$1.27 \cdot 10^{3}$ $(3.4 \cdot 10^{1})$	$1.04 \cdot 10^{3}$ $(2.3 \cdot 10^{1})$
$1.25{\cdot}10^6$	$1.13 \cdot 10^{3}$ $(6.2 \cdot 10^{0})$	$1.49 \cdot 10^{3}$ $(3.1 \cdot 10^{1})$	$1.62 \cdot 10^{3}$ $(3.8 \cdot 10^{1})$	$1.32 \cdot 10^{3}$ $(2.8 \cdot 10^{1})$
$1.5{\cdot}10^6$	$1.38 \cdot 10^{3}$ $(8.2 \cdot 10^{0})$	$1.80 \cdot 10^{3}$ $(3.6 \cdot 10^{1})$	$1.96 \cdot 10^{3}$ $(4.6 \cdot 10^{1})$	$1.59 \cdot 10^{3}$ $(3.1 \cdot 10^{1})$
$1.75{\cdot}10^6$	$1.62 \cdot 10^{3}$ $(1.3 \cdot 10^{1})$	$2.12 \cdot 10^{3}$ $(4.7 \cdot 10^{1})$	$2.32 \cdot 10^{3}$ $(4.5 \cdot 10^{1})$	$1.87 \cdot 10^{3}$ $(4.1 \cdot 10^{1})$
$2{\cdot}10^6$	$1.87 \cdot 10^{3}$ $(1.4 \cdot 10^{1})$	$2.45 \cdot 10^{3}$ $(5.9 \cdot 10^{1})$	$2.66 \cdot 10^{3}$ $(5.3 \cdot 10^{1})$	$2.16 \cdot 10^{3}$ $(5.2 \cdot 10^{1})$

Table 15: Raw runtime data for Java in interpretive mode underlying the plots in Figure 6. Every entry gives the mean of the runs on 100 lists of this size and the corresponding sample standard deviation in parentheses. All times are given in milliseconds.

tuned variant of Algorithm 8, whereas the Standard Template Library of C++ ships an implementation of *Introsort*. Introsort is a decently modified Quicksort with median of three, which falls back to Heapsort whenever the recursion depth of a sublist exceeds $2 \log_2 n$. Thereby it eliminates the quadratic worst case. Introsort was proposed in [Mus97].

The resulting running times are shown in Figures 5, 6 and 7. Each of these figures contains two plots, one comparing the absolute runtime and one with normalized times. The first is suitable to get an overall view of the differences in running time and also allows to directly compare implementations of the same algorithm on different runtime platforms. The normalized plot clearly conveys the relative ranking of the algorithms on a given runtime platform. The three smallest sizes $n \in \{10, 100, 1000\}$ are omitted in the plots, as they would be placed above each other.

The overall results are astonishing: Excluding the highly tuned runtime library versions, dual pivot Quicksort with Yaroslavskiy's partitioning is the by far the fastest of all considered Quicksort variants in both the Java and C++ implementation! This observation clearly conflicts with the result in Table 12 on page 134, where we found that classic Quicksort incurs by far less costs in both machine cost measures $T_n^{\texttt{MMIX}}$ and T_n^{JVM}. The implication is that even the detailed instruction counting model from Chapter 7 seems to miss some relevant aspect regarding the runtime behavior of Algorithm 8.

Furthermore, we notice that the plain version and the runtime library implementation of Algorithm 8 perform almost equally well. This suggests that the modifications Algorithm 8 has undergone while becoming part of the runtime library have hardly measurable influence on the efficiency for sorting random permutations.

In light of the ongoing disput whether Java can be as efficient as C++, it is interesting to compare the absolute runtimes of the algorithms. The C++ versions of Algorithms 8 and 1 are 6 % respectively 10 % faster than the corresponding Java implementations, which is a rather small difference. For Algorithms 7 and 9, the Java versions need 50 % more time. One might suspect that the C++ compiler found a good optimization, which the just-in-time compiler of the JVM missed.

Finally, I included Java `-Xint` in the hope that its runtimes resemble the results for T_n^{JVM} from Chapter 7. Indeed, Algorithm 1 — which needs by far the least number of Bytecode instructions — is the fastest of the considered algorithms on this runtime platform. For the other algorithms, the results are somewhat peculiar: Algorithm 8 is the worst of all, whereas Algorithm 7 surprisingly comes second place. It seems that the model of counting executed Bytecode instructions is too coarse even for Java in interpretive mode.

For the C++ implementations, the high accuracy timer allows to measure single sorting runs. Hence, the noise in measurement gets reduced as n grows. This is clearly visible in the normalized plot of Figure 6. For the Java implementations, the low accuracy of available timers forced me to repeat sorting of a single list anyway. As I used more repetitions for short lists there, noise is equally low for all n.

8.3 Discussion

In this chapter, I conducted a runtime study of all Quicksort variants considered in this thesis. Despite incurring higher cost than classic Quicksort in the primitive instruction cost model of Chapter 7, dual pivot Quicksort with Yaroslavskiy's partitioning method was the fastest sorting method. This confirms the results of earlier runtime tests for tuned variants of the algorithms.

Even though I studied rather detailed cost models in this thesis, the analysis seems still to be too coarse to explain the success of Yaroslavskiy's algorithm. This calls for further investigation. Maybe one has to include some more aspects of modern processors, like caching or pipelining.

9 Pivot Sampling

> “ *See first, think later, then test. But always see first. Otherwise you will only see what you were expecting.* ”
> — Douglas Adams

In the preceding chapters, we have analyzed several dual pivot Quicksorts in great detail. The goal there was to understand the differences in performance of the *basic partitioning methods* — not to come up with an “ultimate Quicksort”. However, one might object that it is somewhat quixotic to ignore tricks that have been successfully used to speed up the basic algorithm for decades. As we will see in this chapter, it pays to first analyze the basic methods, though: Well-understood improvements of classic Quicksort need not apply in the same way to dual pivot variants. Separately studying reveals such interdependencies.

In this chapter, we consider the arguably most successful variation of Quicksort: pivot sampling. Section 3.4.1 gave an overview of proposed selection strategies and cited analyses thereof. It is particularly noteworthy that even with the simplest of those, median of three, classic Quicksort needs only $1.\overline{714285} n \ln n + \mathcal{O}(n)$ comparisons to sort a random permutation of size n [SF96, Theorem 3.5]. This is significantly less than the corresponding expected numbers for both Algorithms 8 and 9, so it will be interesting to see whether dual pivot Quicksort can compete with that.

For dual pivot Quicksort, we have to choose *two* pivots — big surprise. The natural generalization of choosing the single median of k is then to choose the two *tertiles* of k, i. e. the two elements such that there is the same number of elements smaller, between and larger than the pivots. However in this chapter, I consider a more general pivot selection scheme, which allows *arbitrary order statistics of a fixed size sample*. It contains the “tertiles of k” as special case.

The pivot selection scheme is characterized by the non-negative integer constants k, t_1, t_2 and t_3 with $k = t_1 + t_2 + t_3 + 2$.[23] In each partitioning step, we choose a sample of k elements from the list. Let $s_1 < s_2 < \cdots < s_k$ be the sample elements in ascending order. Then, we pick s_{t_1+1} and $s_{t_1+t_2+2}$ as pivots such that t_1 elements are smaller than both pivot, t_2 lie in between and t_3 elements are larger than both:

$$\underbrace{s_1 \cdots s_{t_1}}_{t_1 \text{ elements}} \quad s_{t_1+1} \quad \underbrace{s_{t_1+2} \cdots s_{t_1+t_2+1}}_{t_2 \text{ elements}} \quad s_{t_1+t_2+2} \quad \underbrace{s_{t_1+t_2+3} \cdots s_k}_{t_3 \text{ elements}} \; .$$

If $k = 3t + 2$, then $t_1 = t_2 = t_3$ yields the exact tertiles of the sample as pivots.

[23] Of course, we can always express one of the parameter via the other, e. g. once k, t_1 and t_2 are chosen, $t_3 = k - 2 - t_1 - t_2$ is fully determined. For concise notation, though, it is convenient to have all of them defined.

9.1 Approach of Analysis

Let us start with the probability $\mathrm{P}_{p,q}^{t_1,t_2,t_3}$ of selecting pivots p and q out of a random permutation of $[n]$:

$$\mathrm{P}_{p,q}^{t_1,t_2,t_3} := \frac{\binom{p-1}{t_1}\binom{q-p-1}{t_2}\binom{n-q}{t_3}}{\binom{n}{k}} \,. \tag{9.1}$$

The proof is a simple combinatorial argument: In total, there are $\binom{n}{k}$ different samples. A sample induces pivots p and q iff there are t_1 elements smaller than p, t_2 ones between the pivots and t_3 elements larger than q in the sample. In total, there are $p-1$, $q-p-1$ and $n-q$ small, medium and large elements, respectively. So, a 'good' sample chooses t_1 of the $p-1$ small elements, t_2 of the $q-p-1$ medium elements and t_3 from the set of $n-q$ large elements. Counting the number of choices for such a good sample, divided by the number of all samples, gives eq. (9.1).

The recurrence relation for costs of dual pivot Quicksort derived in Section 4.2 becomes

$$C_n = pc_n + \sum_{1\leqslant p<q\leqslant n} \mathrm{P}_{p,q}^{t_1,t_2,t_3}\left(C_{p-1} + C_{q-p-1} + C_{n-q}\right) \,. \tag{9.2}$$

The solution techniques used in Section 4.2.1 are not directly applicable. There, we used symmetry between the three recursive cost contributions C_{p-1}, C_{q-p-1} and C_{n-q}. With $t_i \neq t_j$, this symmetry is gone.

However, Hennequin's generating function approach can be generalized to asymmetric probabilities $\mathrm{P}_{p,q}^{t_1,t_2,t_3}$. The symbolic description of the weighted class $C(\mathcal{S})$ given as eq. (3.7) on page 47 simply becomes

$$\begin{aligned} C(\mathcal{S}) = \quad & T_M\Big[\circ_k\big(\Delta_{t_1}\big(C(\mathcal{S})\big) \star \Delta_{t_2}(\mathcal{S}) \star \Delta_{t_3}(\mathcal{S})\big)\Big] \\ & + T_M\Big[\circ_k\big(\Delta_{t_1}(\mathcal{S}) \star \Delta_{t_2}\big(C(\mathcal{S})\big) \star \Delta_{t_3}(\mathcal{S})\big)\Big] \\ & + T_M\Big[\circ_k\big(\Delta_{t_1}(\mathcal{S}) \star \Delta_{t_2}(\mathcal{S}) \star \Delta_{t_3}\big(C(\mathcal{S})\big)\big)\Big] \\ & + T_M\big[PC(\mathcal{S})\big] + R_M\big[C_{SLS}(\mathcal{S})\big] \,. \end{aligned}$$

The probability $\mathrm{P}_{p,q}^{t_1,t_2,t_3}$ enters $C(\mathcal{S})$ implicitly via Δ_{t_i} and $\circ_k$ and the sizes of the corresponding subpermutations. The symbolic description translates to a differential equation for $C(z)$:

$$\begin{aligned} C^{(k)}(z) = \quad & T_{M-k}\left[k!\,\frac{C^{(t_1)}(z)}{t_1!}\,\frac{S^{(t_2)}(z)}{t_2!}\,\frac{S^{(t_3)}(z)}{t_3!}\right] \\ & + T_{M-k}\left[k!\,\frac{S^{(t_1)}(z)}{t_1!}\,\frac{C^{(t_2)}(z)}{t_2!}\,\frac{S^{(t_3)}(z)}{t_3!}\right] \\ & + T_{M-k}\left[k!\,\frac{S^{(t_1)}(z)}{t_1!}\,\frac{S^{(t_2)}(z)}{t_2!}\,\frac{C^{(t_3)}(z)}{t_3!}\right] \\ & + T_{M-k}\big[PC^{(k)}(z)\big] + R_{M-k}\big[{C_{SLS}}^{(k)}(z)\big] \\ & \text{with } C_{SLS}(z) = \sum_{\sigma\in\mathcal{S}} C_{SLS}(\sigma)\frac{z^{|\sigma|}}{|\sigma|!} \,. \end{aligned}$$

For given fixed parameters k and t_i, this equation can be explicitly solved using essentially the same approach as in Section 4.2.2. To obtain a general closed solution, Hennequin pursues the more implicit method outlined in Section 3.5.3. In [Hen91, Proposition III.9], he gives the coefficient of the leading $n \ln n$ term for the total costs C_n if the partitioning costs pc_n are linear: For $pc_n = a \cdot n + O(1)$, the solution to eq. (9.2) satisfies

$$C_n = a \cdot g(k, t_1, t_2, t_3) \cdot n \ln n + \mathcal{O}(n) , \tag{9.3}$$

where $g(k, t_1, t_2, t_3)$ is a constant only depending on the parameters k and t_i:

$$g(k, t_1, t_2, t_3) = \left(\mathcal{H}_{k+1} - \sum_{i=1}^{3} \frac{t_i + 1}{k+1} \mathcal{H}_{t_i+1} \right)^{-1} .$$

Thanks to Hennequin's foresighted analysis, we only need to determine the constant a for the costs we are interested in. The expected number of elementary operations needed in the first partitioning step of Algorithms 8 and 9 depends on the pivot probabilities. Nevertheless, we can reuse much of the analyses done in Chapters 4 and 5. There, we derived $\mathbb{E}\left[pc_n \mid p, q\right]$ as an intermediary step and then computed the total expectation as

$$\mathbb{E}\, pc_n = \tfrac{2}{n(n-1)} \sum_{1 \leqslant p < q \leqslant n} \mathbb{E}\left[pc_n \mid p, q\right] .$$

For the pivot selection with parameters k and t_i this changes to

$$\begin{aligned} \mathbb{E}\, pc_n &= \sum_{1 \leqslant p < q \leqslant n} \mathrm{P}_{p,q}^{t_1,t_2,t_3} \cdot \mathbb{E}\left[pc_n \mid p, q\right] \\ &= \tbinom{n}{k}^{-1} \sum_{1 \leqslant p < q \leqslant n} \tbinom{p-1}{t_1}\tbinom{q-p-1}{t_2}\tbinom{n-q}{t_3}\, \mathbb{E}\left[pc_n \mid p, q\right] , \end{aligned}$$

so we only have to do the summation step anew. Trivially, if $\mathbb{E}\left[pc_n \mid p, q\right] = f_1(n, p, q) + f_2(n, p, q)$, we can split the sum and evaluate it for f_1 and f_2 separately. By fully expanding the terms, we can reduce the whole computation for Algorithm 8 to computing

$$\sum_{1 \leqslant p < q \leqslant n} \tbinom{p-1}{t_1}\tbinom{q-p-1}{t_2}\tbinom{n-q}{t_3} f(n, p, q) \tag{9.4}$$
$$\text{for } f(n, p, q) \in \left\{1, p, q, p^2, q^2, pq\right\} .$$

For Algorithm 9, we have some more complicated terms — we also need

$$f(n, p, q) \in \left\{ \tfrac{(q-p-1)p}{p+n-q}, \tfrac{(p-1)^2}{p-1+n-q}, \tfrac{(n-q)^2}{p-1+n-q} \right\} . \tag{9.5}$$

Of course, the latter can be further decomposed by expanding the numerator, what I omitted for conciseness.

These double sums are far from trivial and I could not come up with general closed form. However, for given small values of k and the t_i, Mathematica is able to find a closed form of them. All these closed forms have the form $a \cdot n + O(1)$ required for Hennequin's proposition, so we can indeed give the precise leading term of the costs for small sample sizes.

From a theoretical point of view, these results are unsatisfactorily incomplete. From a practical point of view, only moderate sample sizes will be interesting[24]: We somehow have to determine the order statistics of a sample in each partitioning step. This only contributes to the linear term of the costs as there are $\Theta(n)$ partitioning steps and the sample size is assumed constant, $k = \Theta(1)$ as $n \to \infty$. Therefore, dealing with the sample is asymptotically dominated by swaps and comparisons during the partitioning process. For moderately sized inputs, however, it is far from negligible, so we should keep k fairly small, as well.

Having settled for computer algebra anyway, we can just as well do things in a big way. Therefore, for every tractable k, I compute (the leading term of) the expected number of swaps and comparisons for every possible triple (t_1, t_2, t_3). While we're at it, I also compute the leading term of the number of executed Bytecodes for the corresponding implementations from Section 7.2.

9.2 Results

9.2.1 Algorithm 1 with Pivot Sampling

Of course, for classic Quicksort, we are not restricted to selecting the median, as well. The trivial adaption of the pivot selection scheme described above for dual pivot Quicksort says: For sample size k, we fix t_1 and t_2 such that $k = t_1 + t_2 + 1$ and then select the $(t_1 + 1)$-st element as pivot.

It has been well-known that for classic Quicksort, pivot sampling can greatly reduce the number of needed comparisons. The optimum w. r. t. to the number of comparisons among the possible order statistics of a fixed sample size is always to choose the sample median. In fact, a much stronger result holds. Assume that we choose a *random* t_1 according to some fixed distribution in every partitioning step and then choose the $(t_1 + 1)$-st element from the k-sample. Then, my restricted pivot sampling strategy corresponds to the probability distribution that puts all mass on one value for t_1. In [Sed75, Theorem 8.1], Sedgewick shows that among all those random pivot selection schemes, deterministically selecting the median minimizes the overall number of comparisons. Hennequin generalizes this result to Algorithm 5 in [Hen91, Proposition III.10].

At the same time, the median constitutes the *worst case* among all order statistics when it comes to swaps! Figure 8 shows this situation graphically. Intuitively, the influence of t_1 on the number of swaps can be explained as follows: Any exchanges in Algorithm 1 are done for pairs of elements which are *both* not located in the correct partition. If now the pivot is near the smallest or largest values, there are much less candidates of such "both-out-of-order pairs". Hence, we get less swaps than for a pivot near the median.

[24] These practical considerations serve as nice apologies for my inability to find closed forms of the sums above, don't they?

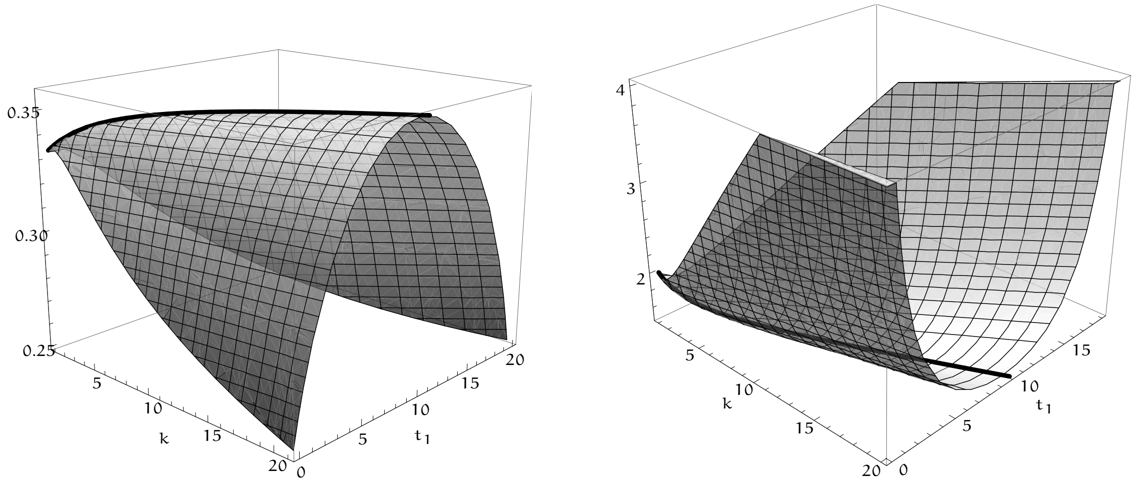


Figure 8: The coefficient of the leading $n \ln n$ term of the number of swaps (left) and comparison (right) for Algorithm 1 with pivot sampling. The parameters k and t_1 are shown on the axes, t_2 is determined as $t_2 = k - 1 - t_1$. The fat line shows the median selection $t_1 = t_2 = \frac{1}{2}(k-1)$. It is clearly visible that for each fixed k the median line corresponds to the minimum in t_1-direction for comparisons, but to the maximum for swaps.

This obvious conflict has not received much attention in the literature. One exception is the nice paper [MR01]. Therein, Martínez et al. determine the optimal sample size and order statistic for the overall cost of Quicksort defined as $\mathtt{C} + \xi \cdot \mathtt{S}$. They find that for $\xi < 10.35$, the optimal order statistic is the median, independent of the sample size k. For my implementations of classic Quicksort, Section 7.4.1 determined the relative runtime contributions of swaps to be $\xi \leqslant 3$. Given that classic Quicksort needs 6 times as many comparisons as swaps, the tradeoff between minimizing the number of comparisons respectively swaps is totally dominated by comparisons in practical implementations. This justifies the lack of interest in the influence of pivot selection on swaps.

9.2.2 Algorithm 8 with Pivot Sampling

As described in Section 9.1, I let Mathematica simplify eq. (9.4) to obtain the expected partitioning costs and then, use Hennequin's result eq. (9.3) to compute the leading term of the overall expected costs. I do this for some small values for k, all possible triples (t_1, t_2, t_3) and the three cost measures: total number of comparisons $\mathtt{C}$, swaps $\mathtt{S}$ and executed Bytecode instructions T_n^{JVM}.

This amounts to quite a bunch of numbers. Therefore I present them in graphical form — as a series of three-dimensional plots (Figures 9, 10 and 11). Each figure shows three plots, one for each of the three considered cost measures. In such a group of three plots, the sample size k is kept constant. I chose $k = 5$, $k = 8$ and $k = 11$ for Figures 9, 10 and 11, respectively.

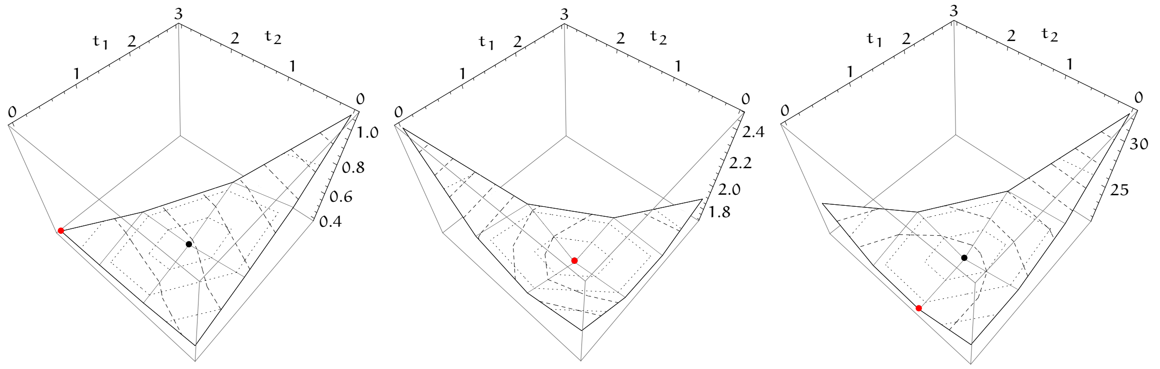


Figure 9: The coefficient of the leading $n \ln n$ term of the total number of swaps (left), comparisons (middle) and executed Bytecode instructions (right) used by Algorithm 8 with pivot sampling for $k = 5$. The red dot shows the minima in the plots, they are located at $(t_1, t_2, t_3) = (0, 3, 0)$ for swaps, $(1, 1, 1)$ for comparisons and $(0, 1, 2)$ for Bytecode instructions.

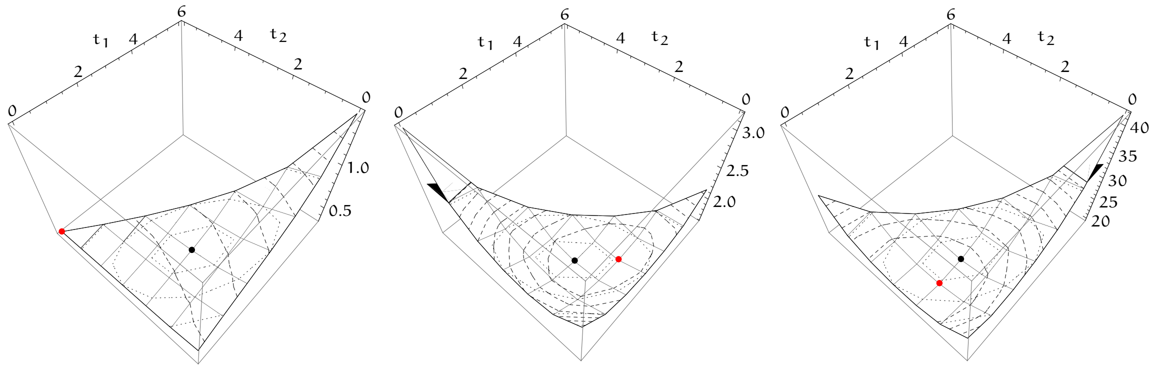


Figure 10: The coefficients of the leading $n \ln n$ terms for Algorithm 8 with pivot sampling with $k = 8$. The minima are located at $(t_1, t_2, t_3) = (0, 6, 0)$ for swaps, $(3, 1, 2)$ for comparisons and $(1, 2, 3)$ for Bytecode instructions.

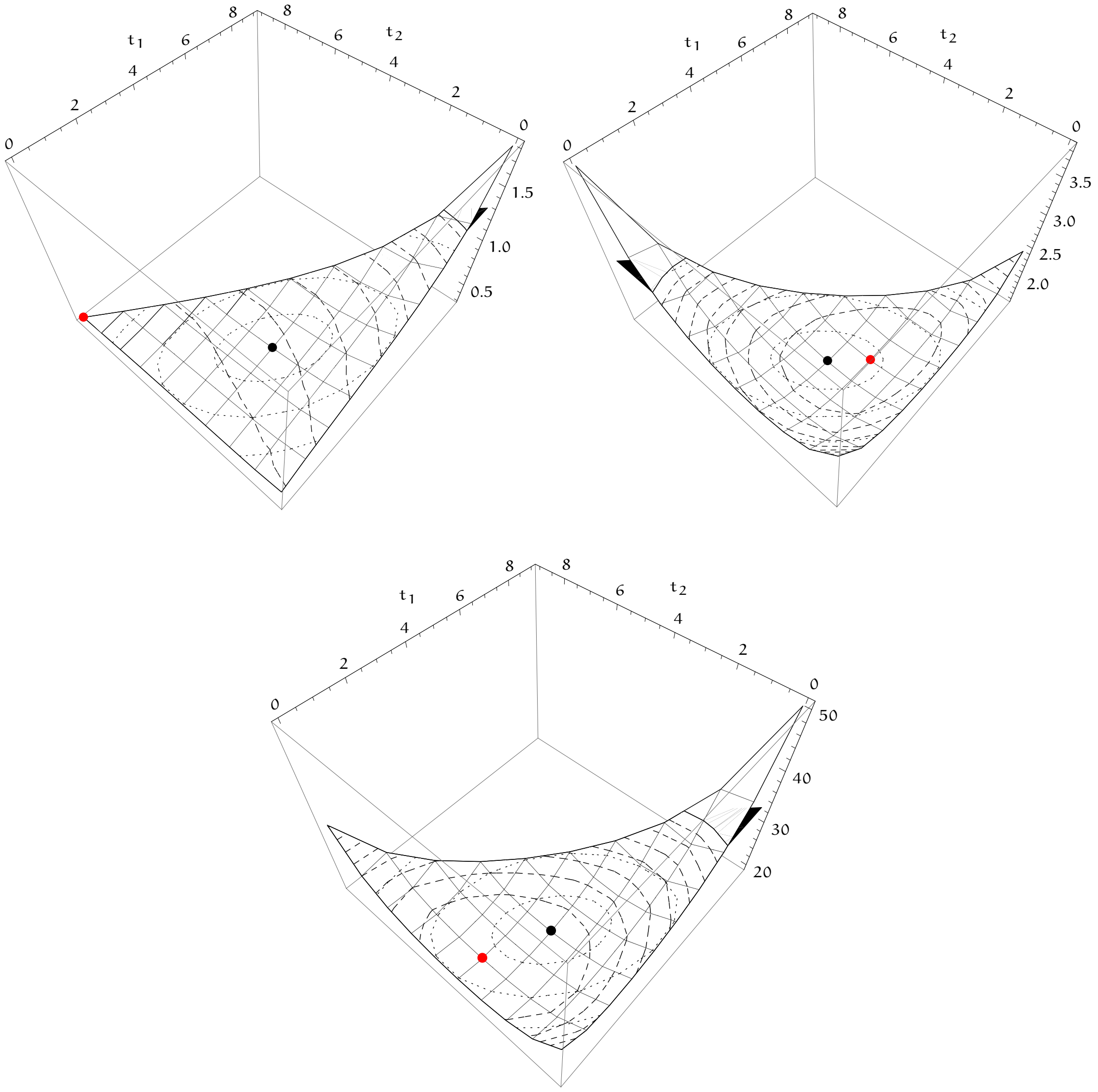


Figure 11: The coefficients of the leading $n \ln n$ terms for Algorithm 8 with pivot sampling with $k = 11$. The upper left plot shows the leading term coefficient for swaps, the upper right the one for comparisons and the lower plot the one for Bytecode instructions. The minima are located at $(t_1, t_2, t_3) = (0, 9, 0)$ for swaps, $(4, 2, 3)$ for comparisons and $(1, 4, 4)$ for Bytecode instructions.

On two axes, t_1 and t_2 are given. Note only values for t_1 and t_2 with $t_1 + t_2 \leqslant k-2$ make sense, therefore the plotted surface has triangular projection on the t_1-t_2-plane. The third dimension directly gives the leading term coefficient. Onto the surface, three types of lines are drawn. The solid line mesh indicates integer values of t_1 respectively t_2. Their projection onto the "ground floor" is a regular grid. These lines allow to easily determine the coordinates of a given point on the surface.

The dashed lines are contour lines of the underlying function, therefore they convey information on the slope of the surface. Finally, the thin dotted lines are contour lines for the distance from the 'middle' point $(t_1, t_2) = (\frac{k-1}{3}, \frac{k-1}{3})$. This point corresponds to choosing the exact tertiles of the sample and it is marked with a black blob. The dotted lines can give a feeling of how asymmetric the pivot selection scheme is, which corresponds to a given point. Finally, in every plot I show the global minimum of the function as a red dot.

Note that in most cases, the minimum is *not* located in the middle. For the number of swaps this does not come as a surprise — see Section 9.2.1 for a discussion. However it is quite uncommon that also for the comparisons, the middle is not optimal — especially as Hennequin showed that equidistantly selected pivots are optimal for Algorithm 5. To understand why Algorithm 8 behaves differently, recall the trick Algorithm 8 uses to save comparisons. In Section 4.4.1 on page 90, we learned that comparison locations that are reached more often, when, say, the second pivot is very large — as line 6 in Algorithm 8 — should first check for small elements, as this comparison is reached often exactly when many small elements exist. Then, chances are better than on average that we do not need a second comparison for this element.

The savings due to this trick are maximal, if the pivots attain their extreme values — and they are minimal for the expected value of the pivots: the tertiles of the whole list. But now, choosing the pivots as tertiles of a sample pushes the pivots further towards this trick's worst case! On the other hand, extreme values for the pivots induce bad recursion trees and therefore greater overall cost. Together, we observe a tradeoff between myopically reducing the costs of the current partitioning step and future savings from lower recursive costs. The result we observe e. g. in the middle plot of Figure 10 is then a compromise between the two extreme interests.

From the arguments adduced up to now one might expect a "ring" of minima around the middle. However, we do not observe such a ring, but rather a very distinct trend in one direction. The detailed analysis of Chapter 4 explains why: m-type elements always cause the second comparison and whereas s-type respectively l-type elements only need the second comparison if they are located in position range $\mathcal{G}$ respectively $\mathcal{K}$. For tertiles as pivots, we have $|\mathcal{K}| \approx 2|\mathcal{G}|$. So, the ranges where s-, m- and l-type elements hurt us roughly have sizes with relative ratio $1 : 3 : 2$. If we only look at minimizing immediate costs of the first partitioning step, we should thus maximize the number of s-type elements — which induces the global worst case!

For a minimal *total* number of comparisons, we cannot go that far. However, this reasoning explains the direction into which we deviate from the middle: The ratio $|S| : |M| :$

$|L|$ of the numbers of small, medium and large elements should be like $3:1:2$ — reciprocal to the sizes of hurting ranges. This nicely matches the minima for comparisons we find: $(t_1, t_2, t_3) = (3, 1, 2)$ for $k = 8$ and $(t_1, t_2, t_3) = (4, 2, 3)$ for $k = 11$.

Looking at the number of swaps a little closer, there is a remarkable asymmetry, as well: Generally, extreme pivot positions are favorable, because then, there are less elements our of order in expectation. However for s-type elements, things are differently in Algorithm 8 — those are *always* swapped, even if they already were in the correct part of the array. To be specific, line 7 can be executed many times with $k = l$, 'swapping' an element with itself.

To minimize the expected number of executed Bytecodes, we can neither afford excessive numbers of comparisons, nor doing too many swaps. This adds another tradeoff facet to our analysis: comparisons favor *many* s-type elements, for swaps, exactly those are the most problematic ones. Considering the last plots of Figures 9, 10 and 11, it is evident that the swaps dominate this tradeoff. When minimizing the number of Bytecodes, we end up with a small value of t_1, which corresponds to less s-type elements than in the uniform case.

Remembering that we have to compute the order statistics in every partitioning step, the smallest values of k are of particular practical interest. Therefore, Table 17 on page 161 contains the optimal choices and their costs also for $k = 3$ and $k = 4$.

For $k = 3$, we get a reduction in the number of executed Bytecodes of 6.3 % — with only two additional comparisons to find and exclude the largest element of the sample. Hence, this might be a quite useful variant in practice. This simple pivot sampling strategy suffices to get rid of 10 % of all swaps and at the same time save 2 % of all comparisons. A reduction in the number of both elementary operations is noteworthy, as for Algorithm 1 the optimal pivot selection lead to an *increase* in the number of swaps.

Selecting the smallest and third smallest elements from a sample of size $k = 4$ requires more effort: By first removing the smallest and then taking the median of the remaining three elements, we already consume 6 comparisons. The reward is an asymptotic reduction of 13.8 % in the number of Bytecodes by saving almost every third swap.

Finally, the case $k = 5$ deserves special attention. For Oracle's Java 7 runtime library, an implementation of Algorithm 8 with *tertiles* of five was chosen. However, Table 17 shows that for random permutations, it would have been better to take the smallest and third largest elements of a sample of five elements as pivots! To quantitatively assess the difference, here are the leading term coefficients of the costs of Algorithm 8 for pivot sampling with $k = 5$ and $(t_1, t_2, t_3) = (1, 1, 1)$ respectively $(t_1, t_2, t_3) = (0, 1, 2)$:

	T_n^{JVM}	C	S
$(1,1,1)$	21.3033	1.7043	0.5514
$(0,1,2)$	20.5769	1.8681	0.4396

The result is remarkable: In the average for large random permutations, we achieve a speedup by 3.5 % over tertiles of five *without any additional cost!*

It must be noted however, that the effect of asymmetric pivot sampling on the efficiency for other input distributions, most notably with non-distinct elements, might be different.

9.2.3 Algorithm 9 with Pivot Sampling

We can compute the leading term coefficients of costs for Algorithm 9, as we did for Algorithm 8 in Section 9.2.2. However, the additional sums needed for this algorithm are more complicated, see eq. (9.5) on page 151. In fact, Mathematica was unable to find closed forms of them when one of the t_i was $\geqslant 6$. For those parameter choices, I cannot offer the corresponding results.

Nevertheless, I prepared the same series of three-dimensional plots as in Section 9.2.2 including all points that could be computed. For $k = 8$, only the corners of the triangle are missing, whereas for $k = 11$, the triangular shape is hardly recognizable and some interesting points are missing, unfortunately. For the detailed description of the features of the plots, I refer the reader to Section 9.2.2.

We observe contrary behavior for swaps and comparisons. For the number of comparisons, it is helpful to have as few m-type elements as possible, as those always induce a second comparison. Regarding swaps, however, m-type elements are the only ones that possibly remain untouched if they are already in the correct range. Small and large elements are always swapped either in line 11 or line 18 in one of the inner loops, or in line 23 of Algorithm 8.

For minimizing the number of executed Bytecode instructions, this difference becomes a tradeoff. As for Algorithm 8, swaps dominate the decision: For all considered k, the minima induce more m-type elements than in the uniform case.

9.3 Discussion

It has been known that pivot sampling can greatly improve Quicksort's performance. For classic Quicksort, many comparisons face rather few comparisons, therefore pivot sampling is best used to reduce the number of comparisons — even at the price of *additional* swaps. This is achieved by taking the median of the sample. The situation only changes, when swaps become much more expensive than comparisons, as shown in [MR01].

Note that the number of comparisons per partitioning is independent of the pivot for Algorithm 1. Hence, the savings in classic Quicksort are *only* due to the *more balanced recursion tree*.

For dual pivot Quicksort, the picture diversifies. First of all, pivot selection affects the recursion tree just as in classic Quicksort. Secondly, Algorithms 8 and 9 need much more swaps than Algorithm 1 — and at the same time save some comparisons. Therefore, it is no longer the utmost goal to save comparisons at any cost. Rather we need to *balance the reduction in both comparisons and swaps*, with some tendency to prefer less swaps.

Thirdly, the algorithms behave asymmetric w. r. t. to the three types of elements — smaller than both pivots, between them or larger than both. For example, medium elements

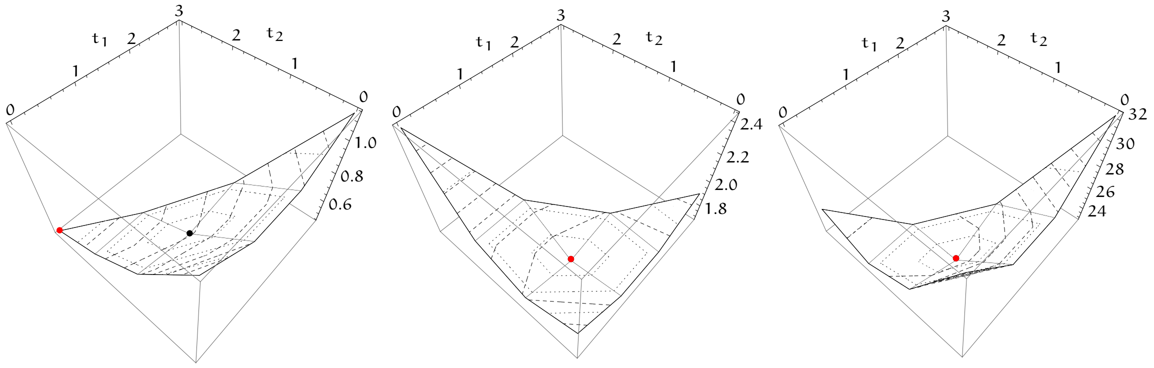


Figure 12: The coefficient of the leading $n \ln n$ term of the total number of swaps (left), comparisons (middle) and executed Bytecode instructions (right) used by Algorithm 9 with pivot sampling for $k = 5$. The red dot shows the minima in the plots, they are located at $(t_1, t_2, t_3) = (0, 3, 0)$ for swaps, $(1, 1, 1)$ for comparisons and $(1, 1, 1)$ for Bytecode instructions.

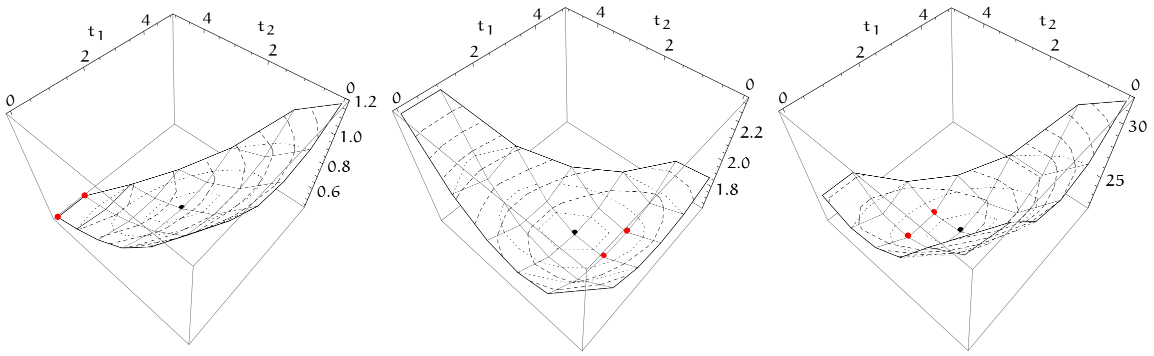


Figure 13: The coefficients of the leading $n \ln n$ terms for Algorithm 9 with pivot sampling with $k = 8$. Notice that some points are missing, namely the extreme points $(t_1, t_2) \in \{(0,0), (0,6), (6,0)\}$. Therefore, the minima need not be global. It seems plausible to assume a monotonic continuation, so that the two minima for swaps are in fact surpassed by $(0, 6, 0)$. The minima for comparisons are located at $(2, 1, 3)$ and $(3, 1, 2)$. For the Bytecode instructions, we have $(1, 3, 2)$ and $(2, 3, 1)$. As those minima lie in the interior of the surface and the edges point upwards, those minima seem safe even without knowing the missing points.

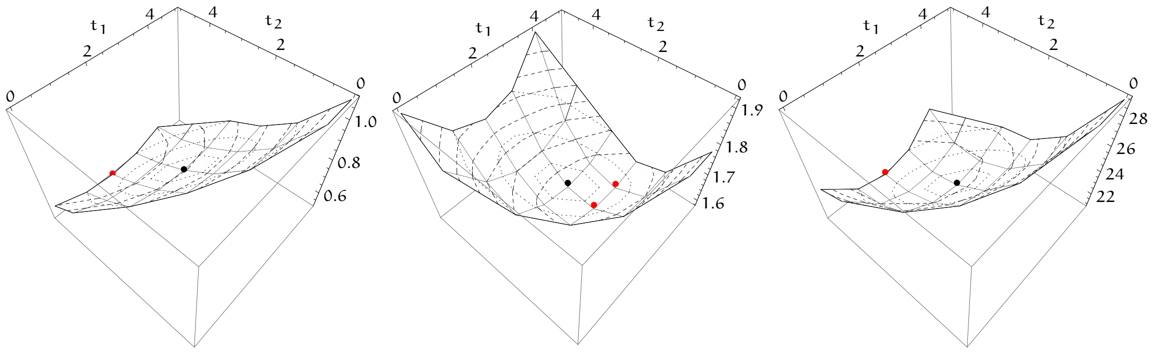


Figure 14: The coefficients of the leading $n \ln n$ terms for Algorithm 9 with pivot sampling with $k = 11$. Notice that quite some points are missing. So, we should not trust the found minima on the rim of the surface. The minima found for the number of comparisons lie in the interior, so they might be the correct optimum. They lie at $(t_1, t_2, t_3) = (3, 2, 4)$ and $(4, 2, 3)$.

	$t = 0$	$t = 1$	$t = 2$	$t = 3$	$t = 4$	$t = 5$
Algorithm 5 with $s = 3$	2	1.754	1.6740	1.6342	1.6105	1.5947
Algorithm 8 tertiles	1.9	1.7043	1.6405	1.6090	1.5903	1.5779
Algorithm 8 best		tertiles	1.6227	1.5849	1.5662	1.5554
Algorithm 9 tertiles	1.8667	1.6842	1.6262	1.5979	1.5812	1.5702
Algorithm 9 best		tertiles	1.6227	1.5849	?	?

Table 16: This table shows the leading term coefficients for the expected number of comparisons for different algorithms with pivot sampling for different sample sizes $k = 3t + 2$.

Algorithm 5 always uses the tertiles of k as pivots. For Algorithms 8 and 9, I give two variants each: The first one chooses the tertiles of the sample, as well. The second variant uses the order statistics, for which the overall number of comparisons is minimized.

	k	(t_1, t_2)	T_n^{JVM}	C	S
Algorithm 1	$s-1=1$	—	18	2	0.3333
	3	$(1,1)$ (median)	16.4571	1.7143	0.3429
	4	$(1,2)$ or $(2,1)$	16.4571	1.7143	0.3429
	5	$(2,2)$ (median)	15.9846	1.6216	0.3475
	8	$(3,4)$ or $(4,3)$	15.7598	1.5760	0.3502
	11	$(5,5)$ (median)	15.5445	1.5309	0.3533

	k	(t_1, t_2, t_3)	T_n^{JVM}	C	S
Algorithm 8	$s-1=2$	—	23.8	1.9	0.6
	3	$(0,0,1)$	22.38	1.86	0.54
	4	$(0,1,1)$	20.9057	1.8868	0.4528
	5	$(0,1,2)$	20.5769	1.8681	0.4396
	8	$(1,2,3)$	19.7279	1.7155	0.4636
	11	$(1,4,4)$	19.3299	1.7941	0.4114

	k	(t_1, t_2, t_3)	T_n^{JVM}	C	S
Algorithm 9	$s-1=2$	—	26	1.8667	0.8
	3	$(0,1,0)$	24	2	0.6
	4	$(0,1,1)$ or $(1,1,0)$	23.7736	1.8113	0.6792
	5	$(1,1,1)$ (tertiles)	22.9474	1.6842	0.7018
	8	$(1,3,2)$ or $(2,3,1)$	21.5593	1.7387	0.5796
	11	$(2,5,2)$[a]	20.9485	1.7829	0.5200

[a] This is potentially not the global minimum, but only the best value that Mathematica was able to compute. In fact, the plot in Figure 14 suggests that t_2 might be even larger at the global minimum.

Table 17: Comparison of the effects of pivot sampling on classic Quicksort and dual pivot Quicksort with Yaroslavskiy's respectively Kciwegdes partitioning. The table shows the leading term coefficients of the expected number of executed Bytecodes, comparisons and swaps for different sample sizes k. The t_i are chosen, such that T_n^{JVM} is minimized. The sample size $k = s - 1$ corresponds to random selection, i .e. no sampling at all.

always need two comparisons to determine their class, whereas for small or large elements, one comparison can suffice. Therefore, part of this asymmetry is inherent to dual pivot Quicksort, not only to my specific implementations. As a consequence, the number of swaps and comparisons of one partitioning step is *not independent of the pivots*.

It is especially interesting to compare the number of comparisons required by asymmetric Algorithms 8 and 9 to what symmetric Algorithm 5 needs. This is done in Table 16 on page 160. For tertiles of k selection, the asymmetry which helps Algorithms 8 and 9 is *reduced*. Yet, the asymmetric algorithms still consume less key comparisons than the symmetric one. If we then even allow them to boost asymmetry by asymmetric order statistics, they can increase their lead even more.

Summarizing, pivot sampling influences the efficiency of dual pivot Quicksorts in *three* competing ways:

- Symmetric pivot selection tends to produce more favorable recursion trees.
- Comparisons generally prefer equidistant pivots, whereas swaps profit from extreme pivot values.
- The inherent asymmetry of dual pivot Quicksorts w. r. t. the equivalence classes of small, medium and large elements invites to use asymmetric pivot sampling to boost helpful asymmetry.

As we have seen for the small sample sizes — where explicit computation of the leading term of the costs was feasible — all three effects have to be considered to explain the optimal order statistics for pivot selection. Numeric results optimizing overall runtime are shown in Table 17.

10 Conclusion

> *I may not have gone where I intended to go, but I think I have ended up where I needed to be.*
>
> — Douglas Adams

In this thesis, I studied the new dual pivot Quicksort variant by Yaroslavskiy (Algorithm 8), which was adopted as default sorting method for Oracle's Java 7 runtime library. I computed its exact expected costs for several cost measures on different levels of abstractions and compared the results with a classic Quicksort variant (Algorithm 1) and older implementations of dual pivot Quicksort. The intriguing result is that the new dual pivot Quicksort can take advantage of asymmetries in the outcomes of key comparisons to save every 20th comparisons. As a byproduct, the analysis of an older dual pivot Quicksort variant due to Sedgewick (Algorithm 7) reveals that this algorithm fails to use asymmetry to its profit. Moreover, the results suggest to reverse comparisons in the algorithm to obtain Algorithm 9, which in fact saves every 15th comparison w. r. t. the ones classic Quicksort uses. [Expected numbers of swaps and comparisons for the different studied algorithms are found in Tables 1, 3, 4 and 5.]

However, the studied dual pivot algorithms need much more swaps than classic Quicksort, which rises the question what dominates in practice. This question was approached by changing to more detailed cost models — including the one used by Knuth in "The Art of Computer Programming". The expected costs under those models invariably favor classic Quicksort over all other variants. In a nutshell, the dual pivot Quicksorts cannot compensate for the many extra swaps they need. [Expected costs for the detailed models are given in Table 12.]

This result is surprising in that Yaroslavskiy's algorithm has shown in extensive empirical studies to be more efficient than previously used sorting methods. To investigate this discrepency, I tried to reproduce a simple runtime study. To be able to directly compare runtime measurements with the results of my analyses and in order to rule out the influence of additional optimizations of Quicksort, I used direct implementations of my algorithms. The expected, but nevertheless puzzling result was that I could reproduce older studies: Yaroslavskiy's algorithm is significantly faster than classic Quicksort and the other dual pivot Quicksorts: The Java implementation of Algorithm 8 saves about $\frac{1}{8}$ of the time of the corresponding classic Quicksort implementation. In C++ the savings amount to $\frac{1}{11}$ of the time for classic Quicksort. [Figures 5 and 6 show the running times.]

Even though I failed to find a conclusive explanation for this success of Yaroslavskiy's new dual pivot Quicksort, the precise analyses and improvements to the algorithms are worth studying on their own. And after all, the fact that classic analysis of algorithms does not fully explain what is going on in YAROSLAVSKIY's algorithm still tells us a lot: Its success in practice is likely to be due to details of modern computer architectures.

Asymmetry

There is one feature that runs like a golden thread through the study of dual pivot Quicksort: Asymmetry. It starts with the obvious asymmetry of Algorithm 8. It does not come as a surprise then, that the different locations where swaps and comparisons happen in Algorithm 8 are executed with different frequencies and that they contribute different costs. However, asymmetry is also found during the analysis of the frequencies: Large pivots contribute differently than small pivots even if the set of sublist sizes agrees. [Frequencies are shown in Table 3, costs contributions in Section 7.4.1.]

Finally, asymmetry reaches its climax in Chapter 9, where we add pivot sampling to Algorithm 8. The optimal order statistics to choose pivots from a sample are highly asymmetric. I showed that this optimum is the result of a delicate tradeoff between several competing extremes. When such asymmetric ingredients add up to a grand total, intuition typically fails. Lucky then, when we are able to base our decisions on firm ground formed by mathematical analysis. [Figures 9, 10 and 11 show examples of these results.]

Typically, humans have the unconditional tendency to prefer symmetry over asymmetry, we consider symmetry æsthetically pleasing and harmonious. This might be a reason why YAROSLAVSKIY's algorithm has not been discovered earlier. From a mathematical point of view, symmetries often *simplify* matters — many short and elegant proofs rely on symmetry arguments. In contrast, many of the arguments in this thesis still give me a feeling of *in*elegance, which is — I guess — the price of asymmetry.

However, symmetries forming a stable state can also mean stagnation. Sometimes it becomes necessary to *break symmetries* to move forward. In the case of dual pivot Quicksort, breaking symmetry allowed to create sorting methods that use less comparisons than we thought would be needed at first sight. [See the "wrong lower bound" from Section 4.4.1.]

10.1 Open Questions

> " *I refuse to answer that question on the grounds that I don't know the answer.* "
>
> — DOUGLAS ADAMS

It is probably at the heart of science that trying to answer questions provokes new questions.[25] Here are some that I stumbled upon during my work on the answers described in this thesis.

[25] I cannot shake off the the resemblance to the Hydra of Lerna, the monster from ancient Greek mythology that grows two new heads for every one that you manage to cut off. On the other hand, it would be an arguably bad heritage to leave our children a world without interesting questions to have a tough time with.

- The first open problem is to identify the mechanisms that make Yaroslavskiy's algorithm so fast in practice. The results of this thesis could not conclusively explain its success, but they might guide the search for reasons in the future — and be it only by indicating where *not* to search.

- I hardly mentioned input distributions other than random permutations, which is the standard model if no additional information about real inputs is available. Its uniformity and symmetry make analysis tractable. However, for a library sorting method, it is not enough to be efficient on random permutations, but it should also reasonably cope with other inputs. Most notably, the presence of equal elements in the list appears frequently in practice. I made some qualitative remarks on the algorithms' behavior in that case, but a thorough treatment as in [Sed77a] for classic Quicksort is in order.

- Except for Chapter 9 on pivot sampling, I confined my study to the most basic implementations of Quicksort. However, many variations of Quicksort are known (see Section 3.4), which might be applied to dual pivot Quicksort. As the example of pivot sampling shows, not all properties simple carry over from classic Quicksort. Hence, studying which of the optimizations of classic Quicksort can be used to improve dual pivot Quicksorts might reveal interesting insights and lead to valuable algorithmic improvements.

- In Chapter 6, I briefly considered the variance of costs of dual pivot Quicksort. For classic Quicksort, variance and higher moments of cost distribution are rather well understood and results for 'symmetric' multi-pivot Quicksort are also available (e. g. [Hen91], [CH01] and [Tan93]). However, the variance of the number of comparisons and swaps of Algorithms 8 and 9 are not yet known.

- Already in the first publication on Quicksort, Hoare noticed that a variation of Quicksort where only one recursive call is executed can be used to select an element by rank [Hoa61b]. This has become known under the name *Quickselect*. Of course, every method to improve Quicksort can also be used to potentially make Quickselect more efficient. It is natural to ask, whether the dual pivot approaches studied in this thesis can be put to a good use in Quickselect, as well.

- Even though I was able to embellish Chapter 9 with these nice three-dimensional plots, it remains unsatisfactorily that no closed form in k, t_1 and t_2 could be obtained for the costs with pivot sampling. For classic Quicksort, [MR01] derives such closed forms, so this might serve as a starting point.

Bibliography

[BLM+91] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith and M. Zagha, **1991**. *A comparison of sorting algorithms for the connection machine CM-2*. In *Annual ACM symposium on Parallel algorithms and architectures*, pages 3–16. ACM Press, New York, USA. ISBN 0-897-91438-4. doi:10.1145/113379.113380.
(Cited on page 33.)

[BM93] J. L. J. Bentley and M. D. McIlroy, **1993**. *Engineering a sort function*. Software: Practice and Experience, 23(11):1249–1265. doi:10.1002/spe.4380231105.
(Cited on pages 13, 34, 36, 37, and 38.)

[BS03] E. Börger and R. F. Stärk, **2003**. *Abstract state machines: a method for high-level system design and analysis*. Springer-Verlag, Berlin. ISBN 3-540-00702-4.
(Cited on page 20.)

[CH01] H.-H. Chern and H.-K. Hwang, **2001**. *Phase changes in random m-ary search trees and generalized quicksort*. Random Structures and Algorithms, 19(3-4):316–358. doi:10.1002/rsa.10005.
(Cited on page 165.)

[CHB06] A. Camesi, J. Hulaas and W. Binder, **2006**. *Continuous Bytecode Instruction Counting for CPU Consumption Estimation*. In *Third International Conference on the Quantitative Evaluation of Systems*, pages 19–30. IEEE. ISBN 0-7695-2665-9. doi:10.1109/QEST.2006.12.
(Cited on page 123.)

[CLRS09] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, **2009**. *Introduction to Algorithms (3rd ed.)*. MIT Press. ISBN 978-0-262-03384-8.
(Cited on pages 11, 28, 34, 35, and 36.)

[CS63] N. Chomsky and M. Schützenberger, **1963**. *The algebraic theory of context-free languages*. Studies in Logic and the Foundations of Mathematics, 35:118–161.
(Cited on page 23.)

[Dur03] M. Durand, **2003**. *Asymptotic analysis of an optimized quicksort algorithm*. Information Processing Letters, 85(2):73–77. doi:10.1016/S0020-0190(02)00351-4.
(Cited on page 34.)

[FM70] W. D. Frazer and A. C. McKellar, **1970**. *Samplesort: A Sampling Approach to Minimal Storage Tree Sorting*. Journal of the ACM, 17(3):496–507. doi:10.1145/321592.321600.
(Cited on pages 32, 33, and 42.)

[FO90] P. Flajolet and A. Odlyzko, **1990**. *Singularity analysis of generating functions*. SIAM Journal on Discrete Mathematics, 3(2):216–240. URL: `http://hal.inria.fr/docs/00/07/57/25/PDF/RR-0826.pdf`.
(Cited on page 49.)

[FS09] P. Flajolet and R. Sedgewick, **2009**. *Analytic Combinatorics*. Cambridge University Press. ISBN 978-0-52-189806-5. URL: `http://algo.inria.fr/flajolet/Publications/AnaCombi/book.pdf`.
(Cited on pages 43, 46, 48, 56, and 61.)

[GKP94] R. L. Graham, D. E. Knuth and O. Patashnik, **1994**. *Concrete mathematics: a foundation for computer science*. Addison-Wesley. ISBN 978-0-20-155802-9.
(Cited on pages 15, 61, and 65.)

[Gre83] D. H. Greene, **1983**. *Labelled formal languages and their uses*. PhD Thesis, Stanford University.
(Cited on pages 33 and 46.)

[Hen89] P. Hennequin, **1989**. *Combinatorial analysis of Quicksort algorithm*. Informatique théorique et applications, 23(3):317–333.
(Cited on pages 33, 43, 48, 56, and 101.)

[Hen91] P. Hennequin, **1991**. *Analyse en moyenne d'algorithmes : tri rapide et arbres de recherche*. PhD Thesis, Ecole Politechnique, Palaiseau.
(Cited on pages 13, 31, 33, 40, 41, 43, 46, 47, 48, 49, 50, 61, 91, 101, 103, 151, 152, and 165.)

[Hoa61a] C. A. R. Hoare, **1961**. *Algorithm 63: Partition*. Communications of the ACM, 4(7):321. doi:10.1145/366622.366642.
(Cited on pages 25 and 26.)

[Hoa61b] C. A. R. Hoare, **1961**. *Algorithm 65: Find*. Communications of the ACM, 4(7):321–322. doi:10.1145/366622.366647.
(Cited on page 165.)

[Hoa62] C. A. R. Hoare, **1962**. *Quicksort*. The Computer Journal, 5(1):10–16. doi:10.1093/comjnl/5.1.10.
(Cited on pages 13, 25, 32, and 39.)

[Inc27] E. L. Ince, **1927**. *Ordinary Differential Equations*. Longmans, Green & Co London. URL: `http://archive.org/stream/ordinarydifferen029666mbp#page/n148/mode/1up`.
(Cited on page 62.)

[Jav09] Java Core Library Development Mailing List, **2009**. *Replacement of Quicksort in java.util.Arrays with new Dual-Pivot Quicksort*. `http://permalink.gmane.org/gmane.comp.java.openjdk.core-libs.devel/2628`. URL: `http://permalink.gmane.org/gmane.comp.java.openjdk.core-libs.devel/2628`.
(Cited on pages 51 and 54.)

[Knu98] D. E. Knuth, **1998**. *The Art Of Computer Programming: Searching and Sorting*, volume 3 of *The Art Of Computer Programming*. Addison Wesley, 2nd edition. ISBN 978-0-20-189685-5.
(Cited on pages 16, 17, 18, 28, 31, 44, 45, 55, and 105.)

[Knu05] D. E. Knuth, **2005**. *The Art of Computer Programming: Volume 1, Fascicle 1. MMIX, A RISC Computer for the New Millennium*. Addison-Wesley. ISBN 0-201-85392-2. URL: `http://www-cs-faculty.stanford.edu/~knuth/fasc1.ps.gz`.
(Cited on pages 108 and 111.)

[Knu11] D. E. Knuth, **2011**. *The Art Of Computer Programming Combinatorial Algorithms, Part1*, volume 4A of *The Art Of Computer Programming*. Addison-Wesley, 1st edition. ISBN 978-0-201-03804-0.
(Cited on pages 44 and 45.)

[KS99] C. Knessl and W. Szpankowski, **1999**. *Quicksort Algorithm Again Revisited*. Discrete Mathematics and Theoretical Computer Science, 3:43 – 64.
(Cited on page 103.)

[LOS09] N. Leischner, V. Osipov and P. Sanders, **2009**. *GPU sample sort*. In *2010 IEEE International Symposium on Parallel Distributed Processing IPDPS*, pages 1–10. IEEE.
(Cited on page 33.)

[LY99] T. Lindholm and F. Yellin, **1999**. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing, Boston, MA, USA, 2nd edition. ISBN 0-201-43294-3.
(Cited on pages 119 and 123.)

[Mar92] C. Martínez Parra, **1992**. *Statistics under the BST model*. PhD Thesis, Univertitat Politècnica de Catalunya. URL: `http://www.lsi.upc.edu/~conrado/research/phd-thesis.ps.gz`.
(Cited on page 45.)

[McG92] C. McGeoch, **1992**. *Analyzing algorithms by simulation: variance reduction techniques and simulation speedups*. ACM Computing Surveys, 24(2):195–212. doi:10.1145/130844.130853.
(Cited on page 140.)

[MN98] M. Matsumoto and T. Nishimura, **1998**. *Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator*. ACM Transactions on Modeling and Computer Simulation, 8(1):3–30.
(Cited on page 140.)

[MR01] C. Martínez and S. Roura, **2001**. *Optimal Sampling Strategies in Quicksort and Quickselect*. SIAM Journal on Computing, 31(3):683. doi:10.1137/S0097539700382108.
(Cited on pages 34, 153, 158, and 165.)

[MT95] C. C. McGeoch and J. D. Tygar, **1995**. *Optimal sampling strategies for quicksort*. Random Structures and Algorithms, 7(4):287–300. doi:10.1002/rsa.3240070403.
(Cited on page 34.)

[Mus97] D. R. Musser, **1997**. *Introspective Sorting and Selection Algorithms*. Software: Practice and Experience, 27(8):983–993.
(Cited on page 147.)

[Par66] R. J. Parikh, **1966**. *On Context-Free Languages*. Journal of the ACM, 13(4):570–581. doi:10.1145/321356.321364.
(Cited on page 23.)

[PWZ96] M. Petkovšek, H. S. Wilf and D. Zeilberger, **1996**. *$A = B$*. A. K. Peters, Wellesley, MA. URL: `http://www.math.upenn.edu/~wilf/Downld.html`.
(Cited on page 59.)

[Sed75] R. Sedgewick, **1975**. *Quicksort*. PhD Thesis, Stanford University.
(Cited on pages 5, 13, 14, 26, 28, 31, 33, 34, 35, 39, 50, 51, 52, 57, 93, and 152.)

[Sed77a] R. Sedgewick, **1977**. *Quicksort with Equal Keys*. SIAM Journal on Computing, 6(2):240–267.
(Cited on page 165.)

[Sed77b] R. Sedgewick, **1977**. *The analysis of Quicksort programs*. Acta Inf., 7(4):327–355. doi:10.1007/BF00289467.
(Cited on pages 28, 30, 33, and 131.)

[Sed78] R. Sedgewick, **1978**. *Implementing Quicksort programs*. Communications of the ACM, 21(10):847–857. doi:10.1145/359619.359631.
(Cited on pages 26 and 35.)

[SF96] R. Sedgewick and P. Flajolet, **1996**. *An Introduction to the Analysis of Algorithms*. Addison-Wesley-Longman. ISBN 978-0-201-40009-0.
(Cited on pages 26, 28, 30, 51, 61, and 149.)

[Sin69] R. C. Singleton, **1969**. *Algorithm 347: an efficient algorithm for sorting with minimal storage [M1]*. Communications of the ACM, 12(3):185–186. doi:10.1145/362875.362901.
(Cited on pages 32, 35, 36, and 39.)

[SW04] P. Sanders and S. Winkel, **2004**. *Super Scalar Sample Sort*. In S. Albers and T. Radzik, editors, *ESA 2004. LNCS, vol. 3221*, pages 784–796. Springer Berlin/Heidelberg. doi:10.1007/978-3-540-30140-0_69.
(Cited on page 33.)

[SW11] R. Sedgewick and K. Wayne, **2011**. *Algorithms*. Addison-Wesley. ISBN 978-0-32-157351-3.
(Cited on pages 37, 107, and 119.)

[Tan93] K.-H. Tan, **1993**. *An asymptotic analysis of the number of comparisons in multipartition quicksort*. PhD thesis, Carnegie Mellon University.
(Cited on pages 33, 41, 91, and 165.)

[van70] M. H. van Emden, **1970**. *Increasing the efficiency of quicksort*. Communications of the ACM, pages 563–567.
(Cited on pages 32 and 34.)

[Wil06] H. S. Wilf, **2006**. *generatingfunctionology*. A K Peters. ISBN 978-1-56-881279-3. URL: `http://www.math.upenn.edu/~wilf/gfologyLinked2.pdf`.
(Cited on page 61.)

[WN12] S. Wild and M. E. Nebel, **2012**. *Average Case Analysis of Java 7's Dual Pivot Quicksort*. In L. Epstein and P. Ferragina, editors, *ESA 2012*, volume LNCS 7501, pages 825–836. Springer Berlin/Heidelberg. URL: `http://wwwagak.cs.uni-kl.de/Veroffentlichungen/Papers/Average-Case-Analysis-of-Java-7s-Dual-Pivot-Quicksort.html`.
(Cited on page 13.)

[Yar09] V. Yaroslavskiy, **2009**. *Dual-Pivot Quicksort*. URL: `http://iaroslavski.narod.ru/quicksort/DualPivotQuicksort.pdf`.
(Cited on page 54.)