Proving JDK's Dual Pivot Quicksort Correct

Bernhard Beckert, Jonas Schiffl, Peter H. Schmitt, and Mattias Ulbrich✉

Karlsruhe Institute of Technology, Germany
✉mattias.ulbrich@kit.edu

Abstract. Sorting is a fundamental functionality in libraries, for which efficiency is crucial. Correctness of the highly optimized implementations is often taken for granted. De Gouw et al. have shown that this certainty is deceptive by revealing a bug in the Java Development Kit (JDK) implementation of TimSort.

We have formally analysed the other implementation of sorting in the JDK standard library: A highly efficient implementation of a dual pivot quicksort algorithm. We were able to deductively prove that the algorithm implementation is correct. However, a loop invariant which is annotated to the source code does not hold.

This paper reports on how an existing piece of non-trivial Java software can be made accessible to deductive verification and successfully proved correct, for which we use the Java verification engine KeY.

1 Introduction

Sorting is an important functionality in every standard library. But implementing sorting efficiently is a non-trivial task. Algorithms found in state-of-the-art runtime libraries are highly optimized for cache-efficient execution on multi-core platforms. De Gouw et al. [6] attempted to prove correctness of the TimSort algorithm, which is used in the Java Development KIT (JDK) for sorting object arrays. In the course of that attempt, they detected a bug in the implementation, attracting a considerable amount of public attention. TimSort is not the only sorting implementation in the JDK: Arrays of primitive data types, such as `int`, `float`, or `char`, are sorted using an implementation of the Dual Pivot Quicksort (DPQS) algorithm [16], a very efficient variation of traditional quicksort. This paper reports on the successful verification of the highly optimized Java routine that implements the DPQS algorithm in the JDK (both oracle's JDK and OpenJDK). It has been proved correct using the deductive verification engine KeY [2]. In this paper, we show how we were able to accommodate the code, which is not at all designed in a verification-friendly fashion, for interactive program verification. The techniques used to make the proof feasible can be transferred to other verification scenarios for real code.

We were able to fully verify correctness of the algorithm and did not find a bug in the code. We found, however, like the authors of the TimSort investigation, that a loop invariant annotated as a comment to the code does actually not hold.

Contributions. We present a mechanised formal proof that the dual pivot quicksort implementation used in the JDK *does* sort an array in ascending order. This consists of proving two properties: The resulting array is sorted, and it is a permutation of the original input. In the course of the verification, lemmas about array permutations have been identified, formulated and proved correct. These can also be used, e.g., for the verification of other permutation-based sorting implementations. Moreover, the paper lists refactoring mechanisms which maintain the semantics of a program but make it more accessible to formal verification.

Structure of the paper. First we present the algorithm and its implementation (Section 2) and the employed technologies – the Java Modeling Language and the KeY system (Section 3). Then, in the main part of the paper, we describe the specification and report on how the program was made more accessible to the KeY tool and how it was proven correct (Section 4). The required effort for the specification and verification is discussed in Section 5. In Section 6, we discuss our discovery of an invariant contained as comment in the implementation that is not always satisfied by the code. We draw conclusions in Section 7.

Related work. We did not find many publications, let alone high profile publications, specifically on formal machine-assisted verification of efficient sorting implementations. But, what we found shows a marked line of development in sorting algorithm verification that parallels the development in the field of program verification in general. Before 2000, subsets or even idealized versions of programming languages were targeted and machine support was mostly restricted to proof checking. The verification of merge sort by Black et al. [3] may serve as an example. A next stage was reached by using an interactive verification system to verify single-thread programs in a real programming language, but written by the people doing the verification. An example is the verification of a counting sort and radix sort program [5]. Another kind of programs written for verification uses a programming language designed for this purpose. As described in a lab report [1], a whole array of sorting programs (selection, bubble, insertion, and quick sort) written in Dafny are proved correct; Leino and Lucio [11] report on the verification of merge sort in Dafny. In the final and challenging stage, programs are verified as they are implemented and employed by a great number of users, as has been done in the much acclaimed paper by de Gouw et al. [6]. One should notice, however, that only normal termination of the TimSort program was analysed.

2 Dual Pivot Quicksort

2.1 The Abstract Algorithm

While the worst-case runtime complexity of comparison-based sorting algorithms is known to be in the class $\mathcal{O}(n \log(n))$, there have been numerous attempts to reduce their "practical" complexity. In 2009, Vladimir Yaroslavskiy [16] suggested a variation of the quicksort algorithm [7] that uses two pivot elements. In conventional quicksort, one element – the pivot – of the array is chosen, and the

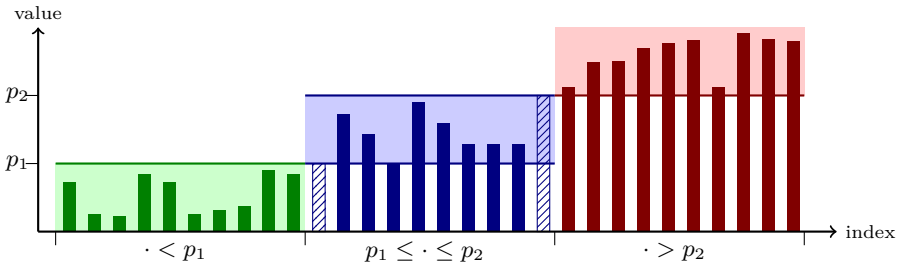

Fig. 1. Illustration of a dual pivot partition

array elements are rearranged into two sections according to how they compare against the pivot. In the dual pivot variant, the partition separates the elements into the three sections according to their comparison against both pivots. Figure 1 exemplarily illustrates the arrangement of the array elements after the partitioning step. The pivot elements are shown as hatched bars. The first part (green in the figure) contains all elements smaller than the smaller pivot element, the middle part (blue) contains all elements between the pivots (inclusively), and the third part (red) consists of all elements greater than the larger pivot. The algorithm proceeds by sorting the three parts recursively by the same principle.

Extensive benchmarking gave empirical evidence that dual pivot sorting performs substantially better on the Java VM than the originally supplied sorting algorithms. This led to the adoption of Yaroslavskiy's Dual Pivot Quicksort implementation as the OpenJDK 7 standard sorting function for primitive data type arrays in 2011. Conclusive explanations for its superior performance appear to be surprisingly hard to find, but evidence points to cache effects [9]. Wild et al. [15] conclude: *"The efficiency of Yaroslavskiy's algorithm in practice is caused by advanced features of modern processors. In models that assign constant cost contributions to single instructions – i.e., locality of memory accesses and instruction pipelining are ignored – classic Quicksort is more efficient."*

2.2 JDK's Implementation

Like many modern programming languages, the standard library of Java uses a portfolio of various sorting algorithms in different contexts. The standard sorting algorithm for object arrays in general is TimSort, an optimized version of merge sort. Calling `sort()` for a primitive data type array, however, leads to the Dual Pivot Quicksort class.

This class, consisting of more than 3000 lines of code, makes use of no less than four different algorithms: Merge sort, insertion sort, counting sort, and quicksort. For the `byte`, `char`, or `short` data types, counting sort is used. Arrays of other primitive data types are sorted using merge sort if they are already partially sorted. For arrays with less than 47 entries, insertion sort is used – in spite of its worse average-case performance – to avoid the comparatively large overhead of quicksort or merge sort.

In all other cases, quicksort is used (e.g., for large integer arrays that are not partially sorted). This "default" option is the subject of our correctness proof. The algorithm itself uses two different partitioning schemes. First, five elements are drawn evenly distributed from the array range and compared; if they are distinct, the range is partitioned with two pivot elements; otherwise, the classical single-pivot three-way partition introduced by Hoare [7] is applied.

The method realising the central part of dual-pivot sorting comprises some 340 lines of Java code containing many optimisations that make the code less comprehensible and more susceptible to oversights. One example of such an optimisation is the approximation of `len / 7` by `(len >> 3) + (len >> 6) + 1`.

Our verification shows that despite these intricacies, the implementation correctly sorts its input. An indication for the fact that the algorithm is difficult to understand is a loop invariant added as a comment to the source code which is not always preserved by the implementation (see Sect. 6).

3 Background

3.1 Java Modeling Language

The Java Modeling Language (JML) [10] is a behavioural interface specification language which follows the principle of design by contract. It is the de facto standard language for the formal specification of Java programs. JML specifications are added to the program source code as special comments beginning with `/*@`. An introductory tutorial to JML can be found in [8].

The language possesses several means for structuring data on more abstract levels, but also for a specification close to the implementation. JML allows to specify programs method-modularly, i.e., to describe (and later verify) the behaviour of each method individually. This keeps the complexity of the verification task down. In the design by contract philosophy, the structuring artefact is the *method contract* consisting of a precondition (`requires`), a postcondition condition (`ensures`) and a *framing clause* (`assignable`), that describes which part of the memory may be modified by the method's code.

All side-effect-free Java expressions may be used in specifications, in addition to specification-specific constructs like the implication connective `==>`, or the quantifiers `\forall` and `\exists`. The expression (`\forall` T `x;` ϕ`;` ψ) evaluates to true iff ψ is true for all elements of the type T that satisfy ϕ. It is equivalent to (`\forall` T `x;` ϕ `==>` ψ).

Besides the concept of method contracts, loop specifications are particularly important for this verification project. Loop specifications are comprised of a loop invariant (`loop_invariant`), a termination measure or variant (`decreases`), and a loop framing clause (`assignable`). Moreover, JML supports *block contracts*, i.e., the specification of (non-loop) statement blocks with local postconditions (`ensures`).

JML is exception- and termination-aware, and we annotated all methods as "`normal_behavior`" indicating that they are expected to terminate normally when called with a satisfied precondition.

The JML dialect that we consider in this work has a built-in data type `\seq` for finite sequences (of values or object references). We also make use of an

extension to JML that allows marking specification clauses as "free" meaning that they can be assumed to be true without proving them. We used this feature to re-use previously verified program properties without re-proving them.

3.2 The Program Verification System KeY

KeY is an interactive theorem prover for verifying the correctness of Java programs w.r.t. their formal JML specifications. The KeY tool is available at the site `www.key-project.org`; more information may be found in the KeY Book [2].

KeY is designed as an interactive tool but has a powerful built in automatic verification engine which can solve many proof obligations automatically. Moreover, state of the art satisfiability modulo theories (SMT) solvers can be called to discharge verification conditions. User interaction is relevant for the most important decision points within the course of a proof, which include quantifier instantiation, definition expansion, or reconfiguration of the strategy.

KeY uses a program logic called Java Dynamic Logic and a sequent calculus for that logic, which follows the verification principle of symbolic execution.

The JML data type of sequences has its logical counterpart in the theory of sequences in KeY [14]. For the verification of dual-pivot quicksort, the abstract data type of sequences is used to abstract from arrays. This is important for verification, as Java arrays are objects on the heap, which makes them susceptible to effects like aliasing.

KeY usually treats integral primitive data types as unbounded mathematical integers. Yet, it also supports the bounded bitvector semantics of Java using modulo operations and overflow checks, but is less efficient in these modes. In some cases, bitvector-oriented proof obligations for loop-free code can be discharged easily by bounded verification engines, like CBMC [4], that encode the challenge as a SAT problem. Since recently CBMC also supports Java as input language.

4 Specification and Verification

Usually, the first challenge of a verification endeavour is to come up with a suitable and concise specification. Sorting algorithms have the neat property that the top-level specification can be stated very concisely and comprehensibly, which is not the case for specifications in general.

The top-level specification for the `sort` method, which is the top-level method to be verified for our verification task, is shown in Listing 1. This JML specification covers the following aspects of the behaviour of the method `sort`:

(a) On termination, the array is sorted in increasing order (lines 5– 7).
(b) On termination, the array contains a permutation of the initial array content (line 8).
(c) The implementation does not modify any existing memory location except the entries of the array (line 9).
(d) The method always terminates (this is the default for JML if a `diverges` clause has not been specified).

Listing 1. Top-level specification of the `sort()` method

```
class DualPivotQuicksort {
  // ...

  /*@ public normal_behavior
    @    ensures (\forall int i; 0 <= i && i < a.length;
    @             (\forall int j; 0 < j && j < a.length;
    @                i < j ==> a[i] <= a[j]));
    @    ensures \seqPerm(\array2seq(a), \old(\array2seq(a)));
    @    assignable a[*];
    @*/
  void sort(int[] a) { ... }
}
```

(e) The method does not throw an exception. This is implied since the contract is declared `normal_behavior`.

Stability of the sorting algorithm is not a relevant issue here as dual-pivot quicksort is applied only to arrays of primitive values.

More specification constructs have to be provided for the verification task, like invariants and contracts for helper methods. These are "auxiliary" specifications that guide the proof but are not part of the requirement specification.

4.1 Proof Management by Gentle Problem Adaptation

The dual-pivot quicksort implementation to be verified is embedded into the portfolio solver of the JDK that is used to sort primitive values. In order to treat it with the formal verification tool KeY, we have slightly adapted the code to accommodate it to the style of programs that KeY can deal with comfortably.

Most importantly, our work focuses on the verification of the dual pivot implementation. The other sorting schemes from which the JDK's portfolio sorting mechanism may choose were not the main goal of this verification (for insights regarding the other schemes, see Sect. 6).

While we claim that we verified the actual JDK implementation, a few semantics-preserving refactorings were necessary to make the code accessible for verification with KeY. A few of these changes were due to the less-supported Java features, but more often they were needed due to a high complexity of the implementation:

- Single-pivot quicksort and dual-pivot quicksort were treated as separate algorithms and encapsulated in separate classes. Part of the code for constructing the dual-pivot partition swaps all the elements with values equal to the pivots to the sides of the array range to be sorted, resulting in a three-way partition of this range, the structure of this part of the algorithm is very similar to the other two classes, and it was therefore encapsulated it in its own class as well.
- Bit shift operations like `x >>> k` were replaced by a division `x / 2`^{k}, which is semantically equivalent when applied to non-negative values of `x`.
- We extracted various code blocks into new private methods. Local variables became fields of the class. This is a semantics-preserving transformation for the sequential execution case (but it is no longer reentrant).

Listing 2. Specification of the partitioning method `split`

```
1  /*@ normal_behaviour
2    @ requires right - left + 1 > 46 && 0 <= left && right < a.length;
3    @ requires (\exists int x; left < x && x < right; a[x] < pivot1);
4    @ requires (\exists int y; left < y && y < right; a[y] > pivot2);
5    @ requires a[e1] < a[e2] && a[e2] < a[e3] && a[e3] < a[e4] && a[e4] < a[e5];
6    @ requires left < e1 && e1 < e2 && e2 < e3 && e3 < e4 && e4 < e5 && e5 < right;
7    @ requires a[e2] == pivot1 && a[e4] == pivot2;
8    @ requires (\forall int i; 0 <= i && i < left; (\forall int j; left <= j && j < a.length; a[i] <= a[j]));
9    @ requires (\forall int i; 0 <= i && i <= right; (\forall int j; right < j && j < a.length; a[i] <= a[j]));
10   @ ensures (\forall int i; left <= i && i < less-1; a[i] < pivot1);
11   @ ensures a[less-1] == pivot1;
12   @ ensures (\forall int j; less <= j && j <= great; pivot1 <= a[j] && a[j] <= pivot2);
13   @ ensures a[great+1] == pivot2;
14   @ ensures (\forall int l; great+1 < l && l <= right; a[l] > pivot2);
15   @ ensures left < less-1;
16   @ ensures great < right-1;
17   @ ensures (\forall int i; 0 <= i && i < left; (\forall int j; left <= j && j < a.length; a[i] <= a[j]));
18   @ ensures (\forall int i; 0 <= i && i <= right; (\forall int j; right < j && j < a.length; a[i] <= a[j]));
19   @ assignable less, great, a[left..right];
20   @*/
21 private static void split(int[] a, int left, int right, int pivot1, int pivot2) {...}
```

Breaking down the code into several methods allowed us to modularize the problem. Besides disentangling the different sorting algorithms, this reduced the complexity of the individual proof obligations.

The points in the code that suggested themselves for method extraction were the partitioning implementation, the initial sorting of the five chosen elements, and several small loops for moving the indices used in the partitioning algorithm.

Besides breaking the problem up into smaller sub-problems, we also reduced complexity by separating three parts of the requirement specification (a) the sortedness property, (b) the permutation property, and (c) the absence of integer overflows. We first proved sortedness, and used sortedness as an assumption for the permutation proof. Overflow checks were disabled for the sortedness and the permutation proof; they were enabled for the last round, which used all previously verified specifications as assumptions.

In order to rely on specifications introduced and proved in an earlier proof iteration as assumptions, "free" specifications were used. Since they were proven correct earlier, reusing them as assumptions was sound.

The absence of implicit exceptions (e.g., division by zero or a null dereference) cannot be switched off in KeY and were thus checked thrice.

4.2 The Sortedness Property

Proving sortedness is quite straight forward by means of assertions on the elements in the array. Quantification and arithmetic over integers are all that is needed. Due to space limitations, we cannot show all intermediate loop invariants and method contracts here, but they can be found on the companion web page. For the reviewers' convenience we added the specification and the sources as appendix A.

Exemplarily, we show in Listing 2 the precondition of the method `split` which implements the partitioning part of the algorithm. The actual partitioning property as illustrated in Figure 1 is covered in lines 10–14. The remainder encodes auxiliary properties needed for the recursive verification to be inductive.

4.3 The Permutation Property

The quicksort algorithm orders an array by continued swapping of its elements. The resulting array is thus easily proved to be a permutation of the original. However, in the optimised Java implementation, the two pivot elements get special treatment: they are exchanged for the boundary elements during the split phase and excluded from the recursive calls. As a consequence, there are intermediate states where it is not obvious that the array after restoring the pivots is a permutation of the initial array.

Let us recall that a sequence b is called a permutation of sequence a iff a and b have the same length n and there is an injective mapping σ from $[0, \ldots, n-1]$ onto itself such that $b[i] = a[\sigma(i)]$ for all $0 \leq i < n$. The mapping σ is called a *witness* of the permutation property. The following lemma helps to deal with the problem described in the previous paragraph (see [13, Corollary 1] for a proof).

Lemma 1. *Let a, b be two arrays such that b is a permutation of a and such that there are two indices i, j with $a[i] = b[i]$ and $a[j] = b[j]$.*

Then there is a witness σ such that $\sigma(i) = i$ and $\sigma(j) = j$.

The lemma was needed to show in KeY that there exists a witness to establish the permutation invariant of the main loop of the partitioning algorithm. The used fixed points are the left- and right-most element of the currently sorted subarray. Only if the pivot values are stored to these places do we obtain a permutation of the original input array.

Already in Listing 1 the function `array2seq` occurred. It transforms Java arrays into mathematical sequences. This allows us to make use of the rich theory of the JML data type `\seq` of finite sequences built into the KeY theorem prover. Lemma 1 and, in fact, a much more general statement have been proved using KeY. Further background material and full proofs of related lemmas and theorems on permutations can also be found in the technical report [13].

4.4 Absence of Integer Overflow

At first, we disabled integer overflow checking in KeY to make the above verification tasks more accessible. After their completion, we marked the already proved specification clauses as free (see Sect. 3.1) to aid in the proof of absence of overflows.

In one case, where index quantiles are computed using bit shift operations, we had to add the precondition that the maximum index in the array is strictly less than `Integer.MAX_VALUE` to an intermediate method. For the verification of this loop-free code with bit shifts, the deductive engine was overwhelmed and the verification condition became extremely large. We employed CBMC [4] to discharge this low-level proof obligation.

4.5 Sorting Pivot Candidates

The pivot elements used by the quicksort algorithms are chosen as quantiles from a set of five elements taken from the array. These five elements are sorted using

an insertion sort algorithm whose loops have been manually unrolled for performance reasons yielding four consecutive nested if-statements. This if-cascade is a real-world example of a piece of code where conventional weakest precondition computation results in exponentially many paths to be considered. We employed manually annotated block contracts to bring complexity back to linear. Alternatively, we could have used KeY's ability to recombine proof goals into one if they represent the same node in the control-flow graph reached on different paths [12].

5 Verification Effort

The first part of the proof consisted of verifying the sortedness property and termination without exceptional behaviour. Including the rather complex process of identifying and implementing suitable adaptations of the problem to a scale where the proof became feasible in KeY, the proof took about two month's time.

Similarly, the permutation proof required some effort outside the actual work with KeY, including the design of the lemma described in Sect. 4.3, its incorporation into the KeY rule set, and improving the proof work-flow by making it possible to use the loop invariants that were already proven in the sortedness proof. Apart from these tasks, the proof of the permutation property required roughly two weeks.

When the sortedness proof finally succeeded, it took 510,439 rule applications, most of them automatic. In total, just under 20 minutes were spent in KeY's automatic mode. In 132 cases, rules had to be applied interactively. 186 of the overall 1892 proof goals were closed by the SMT solver z3.

The proof of the permutation property required more interaction than the sortedness proof, since the rules on sequences and permutation are not usually included in the automatic mode of KeY. The proof of correct permutation of the dual pivot partitioning, which was by far the hardest part, was achieved using a proof script to automate interactive rule applications; the script takes roughly 20 minutes to execute on a machine with a core i7 and 8GB of memory.

6 Invalid Invariant in Single Pivot Quicksort

If there are two or more equal elements among the initially chosen five, the sorting engine resorts to a single pivot partition. While we concentrated on dual-pivot quicksort for the case study reported in this paper, we also verified the implementation of single-pivot quicksort under a slight simplification.[1]

In the course of this verification, we discovered that a loop invariant, stated as a comment in the source code, is not valid. This invariant is attached to the loop for partitioning the array. It states that the array is divided into three parts according to the general quicksort paradigm: (1) elements less than the pivot, (2) elements equal to the pivot, and (3) elements greater than the pivot.

[1] The first element of the array range to be sorted acts as the pivot element, instead of choosing the median of the initially chosen five elements, as in the JDK implementation.

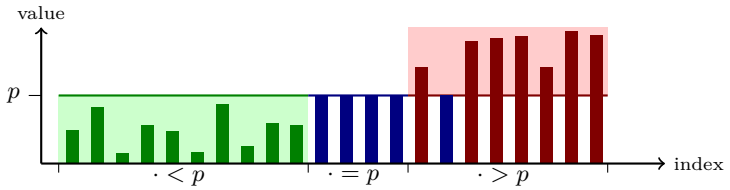

Fig. 2. After a single pivot partition, the array can be in a state where the invariant does not hold as the last part contains one misplaced element at the second position.

Due to the efficient implementation, this invariant may not hold when the partitioning is finished. In many cases, the single pivot partition terminates in a state as shown in Fig. 2, where the last part contains an element that should have been placed in the central part because it is equal to the pivot.

This does not lead to incorrect sorting. The part violating the invariant will itself be sorted recursively, leaving the offending element in the leftmost place, which guarantees a correct order on termination. Since smaller parts of the array with less than 47 elements will eventually be sorted using insertion sort, the violation of the invariant does not persist.

The code can easily been modified such that the violated invariant becomes valid by addition of an extra comparison. A non-representative statistical analysis with random arrays showed that the algorithm is more efficient without the correction.

7 Conclusions

What conclusions can be drawn from this successful verification attempt? First and foremost, it confirms that a real-world optimized algorithm implementation covering 300 lines can be verified using existing verification technology if ...

1. ... *the property to be verified can be concisely specified and formulated.* The specification language and verification technology must possess the right data structures to speak about the program at the right abstraction level – or it must be possible to define them effectively.
2. ... *sensible modularizing refactorings can be made.* In the case of this verification the nature both of the portfolio solver and the sorting algorithm had points at which modularisation was natural.
3. ... *one is willing to add auxiliary specifications.* The modularization introduced by splitting and extraction of individual methods reduces the complexity of individual proof tasks at the price of a considerable amount of intermediate specifications. In the case of KeY, additional user interaction is required to guide the proof. Other tools do this using even more code annotations.

References

1. Abano, C., Chu, G., Eiseman, G., Fu, J., Yu, T.: Lab report, Rutgers University

2. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification: The KeYBook. From Theory to Practice. LNCS 10001, Springer (2016)
3. Black, P.E., Becker, G., Murray, N.V.: Formal verification of a merge sort program with static semantics. ACM SIGPLAN Notices 30(4), 51–60 (April 1995)
4. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS 2988, Springer (2004)
5. de Gouw, S., de Boer, F.S., Rot, J.: Verification of counting sort and radix sort. In: Ahrendt et al. [2], pp. 609–618
6. de Gouw, S., Rot, J., de Boer, F.S., Bubel, R., Hähnle, R.: OpenJDK's Java.utils.Collection.sort() is broken: The good, the bad and the worst case. In: Computer Aided Verification (CAV). LNCS 9206, Springer (2015)
7. Hoare, C.A.R.: Quicksort. The Computer Journal 5(1), 10–16 (1962)
8. Huisman, M., Ahrendt, W., Grahl, D., Hentschel, M.: Formal specification with the Java modeling language. In: Ahrendt et al. [2], pp. 193–241
9. Kushagra, S., López-Ortiz, A., Munro, J.I., Qiao, A.: Multi-pivot quicksort: Theory and experiments. In: Proceedings of the Meeting on Algorithm Engineering & Experiments. pp. 47–60. Society for Industrial and Applied Mathematics (2014)
10. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P., Zimmerman, D.M., Dietl, W.: JML Reference Manual (May 31, 2013), draft Revision 2344
11. Leino, K.R.M., Lucio, P.: An assertional proof of the stability and correctness of natural mergesort. ACM Trans. Comput. Logic 17(1), 6:1–6:22 (Nov 2015)
12. Scheurer, D., Hähnle, R., Bubel, R.: A general lattice model for merging symbolic execution branches. In: International Conference on Formal Engineering Methods (ICFEM). Springer (2016)
13. Schmitt, P.H.: Some notes on permutations. Tech. Rep. 7, Department of Informatics, Karlsruhe Institute of Technology (2017), http://publikationen.bibliothek.kit.edu/1000068624
14. Schmitt, P.H., Bubel, R.: Theories. In: Ahrendt et al. [2], pp. 149–166
15. Wild, S., Nebel, M.E., Neininger, R.: Average case and distributional analysis of Java 7's dual pivot quicksort. CoRR abs/1304.0988 (2013), http://arxiv.org/abs/1304.0988
16. Yaroslavskiy, V.: Dual-pivot quicksort algorithm (2009), http://codeblab.com/wp-content/uploads/2009/09/DualPivotQuicksort.pdf, published online

A Sources

A.1 DualPivotQuicksort.java

This class contains the quicksort implementation with two pivots and all extracted methods.

The specification covers the sortedness and the permutation properties with the clauses containing `\dl_seqPerm` specifying the permutation property. We shortened the symbol names `\dl_seqPerm` and `\dl_array2seq` to `\seqPerm` and `\array2seq` in the main text.

Originally, we composed two separated files since the properties were considered one after the other. The first contained only clauses concerning sortedness, the second contained these clauses annotated as free assumptions and the permutation clauses as proof obligations.

```
class DualPivotQuicksort {

    static int less, great;
    static int e1, e2, e3, e4, e5;

    /*@
      @ normal_behaviour
      @ ensures (\forall int i; 0 <= i && i < a.length; (\forall int j; 0 < j && j < a.length; i <
            ↘  j ==> a[i] <= a[j]));
      @ ensures \dl_seqPerm(\dl_array2seq(a), \old(\dl_array2seq(a)));
      @ assignable less, great, e1,e2,e3,e4,e5,a[*];
      @*/
    static void sort(int[] a) {
        sort(a, 0, a.length-1,true);
    }

    /*@
      @ normal_behaviour
      @ requires 0 <= left && right < a.length;
      @ requires (\forall int i; 0 <= i && i < left; (\forall int j; left <= j && j < a.length; a[
            ↘ i] <= a[j]));
      @ requires (\forall int i; 0 <= i && i <= right; (\forall int j; right < j && j < a.length;
            ↘ a[i] <= a[j]));
      @ ensures (\forall int i; left <= i && i <= right; (\forall int j; left <= j && j <= right;
            ↘ i < j ==> a[i] <= a[j]));
      @ ensures (\forall int i; 0 <= i && i < left; (\forall int j; left <= j && j < a.length; a[i
            ↘ ] <= a[j]));
      @ ensures (\forall int i; 0 <= i && i <= right; (\forall int j; right < j && j < a.length; a
            ↘ [i] <= a[j]));
      @ ensures \dl_seqPerm(\dl_array2seq(a), \old(\dl_array2seq(a)));
      @ assignable less, great, e1,e2,e3,e4,e5, a[left..right];
      @ measured_by right - left + 5;
      @*/
    static void sort(int[] a, int left, int right, boolean leftmost) {
        int length = right - left + 1;
        if (length > 46) {
            prepare_indices(a, left, right);
            if  (allLess(a[e1],a[e2],a[e3],a[e4],a[e5])) {// usually, ThreeWayQs will be done if
                  ↘ two of the pivots are equal

                split(a, left, right, a[e2], a[e4]);

                int lless = less;
                int lgreat = great;
                // Sort left and right parts recursively, excluding known pivots
                sort(a, left, lless - 2, leftmost);
                sort(a, lgreat + 2, right, false);

                // SwapValues.java if central part is large
                // Sort center part recursively
                sort(a, lless, lgreat, false);
            } else {
                insertionSort(a, left, right, leftmost);
            }
        } else if (length > 1) {
            insertionSort(a, left, right, leftmost);
        }
```

```
}

static boolean allLess(int e1, int e2, int e3, int e4, int e5) {
    return e1<e2 && e2<e3 && e3<e4 && e4<e5;
}

/*@
  @ normal_behaviour
  @ requires right - left + 1 > 46 && 0 <= left && right < a.length;
  @ requires (\exists int x; left < x && x < right; a[x] < pivot1); //z.B.a[e1]
  @ requires (\exists int y; left < y && y < right; a[y] > pivot2); //z.B.a[e5]
  @ requires a[e1] < a[e2] && a[e2] < a[e3] && a[e3] < a[e4] && a[e4] < a[e5];
  @ requires left < e1 && e1 < e2 && e2 < e3 && e3 < e4 && e4 < e5 && e5 < right;
  @ requires a[e2] == pivot1 && a[e4] == pivot2;
  @ requires (\forall int i; 0 <= i && i < left; (\forall int j; left <= j && j < a.length; a[
        ↘ i] <= a[j]));
  @ requires (\forall int i; 0 <= i && i <= right; (\forall int j; right < j && j < a.length;
        ↘ a[i] <= a[j]));
  @ ensures (\forall int i; left <= i && i < less-1; a[i] < pivot1);
  @ ensures a[less-1] == pivot1;
  @ ensures (\forall int j; less <= j && j <= great; pivot1 <= a[j] && a[j] <= pivot2);
  @ ensures a[great+1] == pivot2;
  @ ensures (\forall int l; great+1 < l && l <= right; a[l] > pivot2);
  @ ensures left < less-1;
  @ ensures great < right-1;
  @ ensures (\forall int i; 0 <= i && i < left; (\forall int j; left <= j && j < a.length; a[i
        ↘ ] <= a[j]));
  @ ensures (\forall int i; 0 <= i && i <= right; (\forall int j; right < j && j < a.length; a
        ↘ [i] <= a[j]));
  @ ensures \dl_seqPerm(\dl_array2seq(a), \old(\dl_array2seq(a)));
  @ assignable less, great, a[left..right];
  @*/
static void split(int[] a, int left, int right, int pivot1, int pivot2) {
    less  = left;  // The index of the first element of center part
    great = right; // The index before the first element of right part

    /*
     * The first and the last elements to be sorted are moved to the
     * locations formerly occupied by the pivots. When partitioning
     * is complete, the pivots are swapped back into their final
     * positions, and excluded from subsequent sorting.
     */
    a[e2] = a[left];
    a[e4] = a[right];

    move_less_right(a, left, right, pivot1);
    move_great_left(a, left, right, pivot2);
    outer:
    /*@
      @ loop_invariant
      @    less-1 <= k && k <= great+1
      @ && left < less && less <= right
      @ && left <= great && great < right
      @ && (k <= great || less <= k)
      @ && pivot1 < pivot2
      @ && left < less && great < right
      @ && (\forall int i; left < i && i < less; a[i] < pivot1)
      @ && (\forall int j; less <= j && j < k+1 && j <= great; pivot1 <= a[j] && a[j] <=
            ↘ pivot2)
      @ && (\forall int l; great < l && l < right; a[l] > pivot2)
      @ && (\exists int x; left < x && x < right; a[x] < pivot1)
      @ && (\exists int y; left < y && y < right; a[y] > pivot2)
      @ && (\forall int i; 0 <= i && i < left; (\forall int j; left <= j && j < a.length; a[i]
            ↘  <= a[j]))
      @ && (\forall int i; 0 <= i && i <= right; (\forall int j; right < j && j < a.length; a[
            ↘ i] <= a[j]));
      @ loop_invariant \dl_seqPerm(\dl_seqUpd(\dl_seqUpd(\dl_array2seq(a), left, pivot1),
            ↘ right, pivot2), \old(\dl_array2seq(a)));
      @ assignable less, great, a[left..right];
      @ decreases great + 5 - k;
      @
      @*/
    for (int k = less - 1; ++k <= great; ) {
        loop_body(a, k, left, right, pivot1, pivot2);
    }

    // Swap pivots into their final positions
    a[left]  = a[less  - 1]; a[less  - 1] = pivot1;
    a[right] = a[great + 1]; a[great + 1] = pivot2;
```

```
}

/*@
  @ normal_behaviour
  @ requires
  @    less <= k && k <= great
  @ && 0 <= k && k < a.length
  @ && 0 <= left && left < less && less <= right
  @ && left <= great && great < right && right < a.length
  @ && pivot1 < pivot2
  @ && (\forall int i; left < i && i < less; a[i] < pivot1)
  @ && (\forall int j; less <= j && j < k && j <= great; pivot1 <= a[j] && a[j] <= pivot2)
  @ && (\forall int l; great < l && l < right; a[l] > pivot2)
  @ && (\exists int x; left < x && x < right; a[x] < pivot1)
  @ && (\exists int y; left < y && y < right; a[y] > pivot2);
  @ requires (\forall int i; 0 <= i && i < left; (\forall int j; left <= j && j < a.length; a[
        ↘ i] <= a[j]));
  @ requires (\forall int i; 0 <= i && i <= right; (\forall int j; right < j && j < a.length;
        ↘ a[i] <= a[j]));
  @
  @ ensures
  @    (k <= great || less <= k)
  @ && less-1 <= k && k <= great+1
  @ && left <= less && less <= right
  @ && left <= great && great <= right
  @ && pivot1 < pivot2
  @ && (\forall int i; left < i && i < less; a[i] < pivot1)
  @ && (\forall int j; less <= j && j < k+1 && j <= great; pivot1 <= a[j] && a[j] <= pivot2)
  @ && (\forall int l; great < l && l < right; a[l] > pivot2)
  @ && (\exists int x; left < x && x < right; a[x] < pivot1)
  @ && (\exists int y; left < y && y < right; a[y] > pivot2)
  @ && \old(less) <= less
  @ && great <= \old(great);
  @ ensures (\forall int i; 0 <= i && i < left; (\forall int j; left <= j && j < a.length; a[i
        ↘ ] <= a[j]));
  @ ensures (\forall int i; 0 <= i && i <= right; (\forall int j; right < j && j < a.length; a
        ↘ [i] <= a[j]));
  @ ensures \old(a[left]) == a[left];
  @ ensures \old(a[right]) == a[right];
  @ ensures \dl_seqPerm(\dl_array2seq(a), \old(\dl_array2seq(a)));
  @ assignable less, great, a[left..right];
  @*/
static void loop_body(int[] a, int k, int left, int right, int pivot1, int pivot2) {
        int ak = a[k];
        if (ak < pivot1) { // Move a[k] to left part
            a[k] = a[less];
            a[less] = ak;
            ++less;
        } else if (ak > pivot2) { // Move a[k] to right part
            move_great_left_in_loop(a, k, left, right, pivot2);
            if (great == k) {
                great--;
                return;
            }
            if (a[great] < pivot1) { // a[great] <= pivot2
                a[k] = a[less];
                a[less] = a[great];
                ++less;
            } else { // pivot1 <= a[great] <= pivot2
                a[k] = a[great];
            }
            a[great] = ak;
            --great;
        }
    }

/*@
  @ normal_behaviour
  @ requires 0 <= left && left < right && right - left >= 46 && right < a.length;
  @ requires a.length > 46;
  @ requires (\forall int i; 0 <= i && i < left; (\forall int j; left <= j && j < a.length; a[
        ↘ i] <= a[j]));
  @ requires (\forall int i; 0 <= i && i <= right; (\forall int j; right < j && j < a.length;
        ↘ a[i] <= a[j]));
  @ ensures a[e1] <= a[e2] && a[e2] <= a[e3] && a[e3] <= a[e4] && a[e4] <= a[e5];
  @ ensures left < e1 && e1 < e2 && e2 < e3 && e3 < e4 && e4 < e5 && e5 < right;
  @ ensures (\forall int i; 0 <= i && i < left; (\forall int j; left <= j && j < a.length; a[i
        ↘ ] <= a[j]));
```

```
  @ ensures (\forall int i; 0 <= i && i <= right; (\forall int j; right < j && j < a.length; a
      ↘ [i] <= a[j]));
  @ ensures \dl_seqPerm(\dl_array2seq(a), \old(\dl_array2seq(a)));
  @ assignable e1,e2,e3,e4,e5, a[left..right];
  @*/
static void prepare_indices(int[] a, int left, int right) {
    {calcE(left, right);}
    eInsertionSort(a,left,right,e1,e2,e3,e4,e5);
}

/*@
  @ normal_behaviour
  @ requires 0 <= left && right < a.length;
  @ requires left == less && less < great && great < a.length;
  @ requires (\exists int j; less+1 <= j && j < great; a[j] >= pivot1);
  @ ensures less < great;
  @ ensures (\forall int i; left < i && i < less; a[i] < pivot1);
  @ ensures a[less] >= pivot1;
  @ ensures \old(less) < less;
  @ ensures \dl_seqPerm(\dl_array2seq(a), \old(\dl_array2seq(a)));
  @ assignable less;
  @*/
static void  move_less_right(int[] a, int left, int right, int pivot1) {
    /*@
      @ loop_invariant
      @     0 <= less && less <= great && great < a.length
      @  && (\forall int i; left < i && i < less+1; a[i] < pivot1)
      @  && (\exists int j; less+1 <= j && j < great; a[j] >= pivot1)
      @  && \old(less) <= less
      @  && \dl_seqPerm(\dl_array2seq(a), \old(\dl_array2seq(a)));
      @  assignable less;
      @  decreases great - less;
      @*/
    while (a[++less] < pivot1) {
    }
}

/*@
  @ normal_behaviour
  @ requires 0 <= left && left <= less && less < great && great == right && right < a.length;
  @ requires (\exists int i; less <= i && i < great; a[i] <= pivot2);
  @ ensures less <= great;
  @ ensures (\forall int i; great < i && i < right; a[i] > pivot2);
  @ ensures a[great] <= pivot2;
  @ ensures great < \old(great);
  @ ensures \dl_seqPerm(\dl_array2seq(a), \old(\dl_array2seq(a)));
  @ assignable great;
  @*/
static void move_great_left(int[] a, int left, int right, int pivot2) {
  /*@
    @ loop_invariant great > 0;
    @ loop_invariant left <= great && great <= right;
    @ loop_invariant less <= great;
    @ loop_invariant (\forall int i; great-1 < i && i < right; a[i] > pivot2);
    @ loop_invariant (\exists int i; less <= i && i < great; a[i] <= pivot2);
    @ loop_invariant \dl_seqPerm(\dl_array2seq(a), \old(\dl_array2seq(a)));
    @ decreases great;
    @ assignable great;
    @*/
    while (a[--great] > pivot2) {
    }
}

/*@
  @ normal_behaviour
  @ requires 0 <= k && k <= great && great < a.length;
  @ requires (\exists int i; left <= i && i <= great; a[i] <= pivot2);
  @ ensures 0 <= great;
  @ ensures (\forall int i; great < i && i <= \old(great); a[i] > pivot2);
  @ ensures a[great] <= pivot2 || great == k;
  @ ensures k <= great && great <= \old(great);
  @ ensures \dl_seqPerm(\dl_array2seq(a), \old(\dl_array2seq(a)));
  @ assignable great;
  @*/
static void move_great_left_in_loop(int[] a, int k, int left, int right, int pivot2) {
    /*@
      @ loop_invariant
      @     k <= great && 0 <= great
      @  && (\exists int i; left <= i && i <= great; a[i] <= pivot2)
      @  && (\forall int i; great < i && i <= \old(great); a[i] > pivot2)
      @  && great <= \old(great)
```

```
      @  && \dl_seqPerm(\dl_array2seq(a), \old(\dl_array2seq(a)));
      @ decreases great;
      @ assignable great;
      @*/
    while (a[great] > pivot2 && great != k) {
        --great;
    }
}

/*@
  @ normal_behaviour
  @ requires 0 <= left && left < right && right - left >= 46;
  @ ensures left < e1 && e1 < e2 && e2 < e3 && e3 < e4 && e4 < e5 && e5 < right;
  @ assignable e1,e2,e3,e4,e5;
  @*/
static void calcE(int left, int right) {
    int length = right - left + 1;
    int seventh = (length / 8) + (length / 64) + 1;
    e3 = (left + right) / 2; // The midpoint
    e2 = e3 - seventh;
    e1 = e2 - seventh;
    e4 = e3 + seventh;
    e5 = e4 + seventh;
}

/*@
  @ normal_behaviour
  @ requires a.length > 46;
  @ requires 0 <= left && left < e1 && e5 < right && right < a.length;
  @ requires left < e1 && e1 < e2 && e2 < e3 && e3 < e4 && e4 < e5 && e5 < right;
  @ requires (\forall int i; 0 <= i && i < left; (\forall int j; left <= j && j < a.length; a[
      ↘ i] <= a[j]));
  @ requires (\forall int i; 0 <= i && i <= right; (\forall int j; right < j && j < a.length;
      ↘ a[i] <= a[j]));
  @ ensures a[e1] <= a[e2] && a[e2] <= a[e3] && a[e3] <= a[e4] && a[e4] <= a[e5];
  @ ensures (\forall int i; 0 <= i && i < left; (\forall int j; left <= j && j < a.length; a[i
      ↘ ] <= a[j]));
  @ ensures (\forall int i; 0 <= i && i <= right; (\forall int j; right < j && j < a.length; a
      ↘ [i] <= a[j]));
  @ ensures \dl_seqPerm(\dl_array2seq(a), \old(\dl_array2seq(a)));
  @ assignable a[left..right];
  @*/
static void eInsertionSort(int[] a, int left, int right, int e1, int e2, int e3, int e4, int
     ↘ e5) {
    /*@
      @ ensures (a[e1] <= a[e2]);
      @ ensures (\forall int i; 0 <= i && i < left; (\forall int j; left <= j && j < a.length;
          ↘  a[i] <= a[j]));
      @ ensures (\forall int i; 0 <= i && i <= right; (\forall int j; right < j && j < a.
          ↘ length; a[i] <= a[j]));
      @ ensures \dl_seqPerm(\dl_array2seq(a), \old(\dl_array2seq(a)));
      @ assignable a[e1], a[e2];
      @ signals_only \nothing;
      @*/
    {
    if (a[e2] < a[e1]) { int t = a[e2]; a[e2] = a[e1]; a[e1] = t; }
    }

    /*@
      @ ensures (a[e1] <= a[e2] && a[e2] <= a[e3]);
      @ ensures (\forall int i; 0 <= i && i < left; (\forall int j; left <= j && j < a.length;
          ↘  a[i] <= a[j]));
      @ ensures (\forall int i; 0 <= i && i <= right; (\forall int j; right < j && j < a.
          ↘ length; a[i] <= a[j]));
      @ ensures \dl_seqPerm(\dl_array2seq(a), \old(\dl_array2seq(a)));
      @ assignable a[e1], a[e2], a[e3];
      @ signals_only \nothing;
      @*/
    {
        if (a[e3] < a[e2]) { int t = a[e3]; a[e3] = a[e2]; a[e2] = t;
            if (t < a[e1]) { a[e2] = a[e1]; a[e1] = t; }
        }
    }

    /*@
      @ ensures (a[e1] <= a[e2] && a[e2] <= a[e3] && a[e3] <= a[e4]);
      @ ensures (\forall int i; 0 <= i && i < left; (\forall int j; left <= j && j < a.length;
          ↘  a[i] <= a[j]));
      @ ensures (\forall int i; 0 <= i && i <= right; (\forall int j; right < j && j < a.
          ↘ length; a[i] <= a[j]));
      @ ensures \dl_seqPerm(\dl_array2seq(a), \old(\dl_array2seq(a)));
```

```
      @ assignable a[e1], a[e2], a[e3], a[e4];
      @ signals_only \nothing;
      @*/
    {
        if (a[e4] < a[e3]) { int t = a[e4]; a[e4] = a[e3]; a[e3] = t;
            if (t < a[e2]) { a[e3] = a[e2]; a[e2] = t;
                if (t < a[e1]) { a[e2] = a[e1]; a[e1] = t; }
            }
        }
    }

    /*@
      @ ensures (a[e1] <= a[e2] && a[e2] <= a[e3] && a[e3] <= a[e4] && a[e4] <= a[e5]);
      @ ensures (\forall int i; 0 <= i && i < left; (\forall int j; left <= j && j < a.length;
            ↘  a[i] <= a[j]));
      @ ensures (\forall int i; 0 <= i && i <= right; (\forall int j; right < j && j < a.
            ↘ length; a[i] <= a[j]));
      @ ensures \dl_seqPerm(\dl_array2seq(a), \old(\dl_array2seq(a)));
      @ assignable a[e1], a[e2], a[e3], a[e4], a[e5];
      @ signals_only \nothing;
      @*/
    {
        if (a[e5] < a[e4]) { int t = a[e5]; a[e5] = a[e4]; a[e4] = t;
            if (t < a[e3]) { a[e4] = a[e3]; a[e3] = t;
                if (t < a[e2]) { a[e3] = a[e2]; a[e2] = t;
                    if (t < a[e1]) { a[e2] = a[e1]; a[e1] = t; }
                }
            }
        }
    }
}


/*@
  @ normal_behaviour
  @ requires 0 <= left && right < a.length;
  @ requires (\forall int i; 0 <= i && i < left; (\forall int j; left <= j && j < a.length; a[
        ↘ i] <= a[j]));
  @ requires (\forall int i; 0 <= i && i <= right; (\forall int j; right < j && j < a.length;
        ↘ a[i] <= a[j]));
  @ ensures (\forall int i; left <= i && i <= right; (\forall int j; left <= j && j <= right;
        ↘ i < j ==> a[i] <= a[j]));
  @ ensures (\forall int i; 0 <= i && i < left; (\forall int j; left <= j && j < a.length; a[i
        ↘ ] <= a[j]));
  @ ensures (\forall int i; 0 <= i && i <= right; (\forall int j; right < j && j < a.length; a
        ↘ [i] <= a[j]));
  @ ensures \dl_seqPerm(\dl_array2seq(a), \old(\dl_array2seq(a)));
  @ assignable a[left..right];
  @*/
static void insertionSort(int[] a, int left, int right, boolean leftmost) {
    if (leftmost) {
        /*
         * Traditional (without sentinel) insertion sort,
         * optimized for server VM, is used in case of
         * the leftmost part.
         */
        for (int i = left, j = i; i < right; j = ++i) {
            int ai = a[i + 1];
            while (ai < a[j]) {
                a[j + 1] = a[j];
                if (j-- == left) {
                    break;
                }
            }
            a[j + 1] = ai;
        }
    } else {
        /*
         * Skip the longest ascending sequence.
         */
        do {
            if (left >= right) {
                return;
            }
        } while (a[++left] >= a[left - 1]);

        /*
         * Every element from adjoining part plays the role
         * of sentinel, therefore this allows us to avoid the
         * left range check on each iteration. Moreover, we use
```

```
             * the more optimized algorithm, so called pair insertion
             * sort, which is faster (in the context of Quicksort)
             * than traditional implementation of insertion sort.
             */
            for (int k = left; ++left <= right; k = ++left) {
                int a1 = a[k], a2 = a[left];

                if (a1 < a2) {
                    a2 = a1; a1 = a[left];
                }
                while (a1 < a[--k]) {
                    a[k + 2] = a[k];
                }
                a[++k + 1] = a1;

                while (a2 < a[--k]) {
                    a[k + 1] = a[k];
                }
                a[k + 1] = a2;
            }
            int last = a[right];

            while (last < a[--right]) {
                a[right + 1] = a[right];
            }
            a[right + 1] = last;
        }
        return;
    }
}
```

A.2 ThreeWayQs.java

This class contains the single pivot quicksort implementation.

```
//import java.util.Arrays;
//import java.util.Random;

class ThreeWayQs {

    private static int less;
    private static int great;
    private static int pivot;

    /*@
      @ public normal_behaviour
      @ assignable less, great, pivot, a[*];
      @ ensures (\forall int i; 0 <= i && i < a.length; (\forall int j; 0 <= j && j < a.length; i
          ↘ < j ==> a[i] <= a[j]));
      @*/
    public static void sort(int[] a) {
        if (a.length > 0) {
            sort(a, 0, a.length-1);
        }
    }

    /*@
      @ normal_behaviour
      @ requires 0 <= left && right < a.length;
      @ requires (\forall int i; 0 <= i && i < left; (\forall int j; left <= j && j < a.length; a[
          ↘ i] <= a[j]));
      @ requires (\forall int i; 0 <= i && i <= right; (\forall int j; right < j && j < a.length;
          ↘ a[i] <= a[j]));
      @ ensures (\forall int i; left <= i && i <= right; (\forall int j; left <= j && j <= right;
          ↘ i < j ==> a[i] <= a[j]));
      @ ensures (\forall int i; 0 <= i && i < left; (\forall int j; left <= j && j < a.length; a[i
          ↘ ] <= a[j]));
      @ ensures (\forall int i; 0 <= i && i <= right; (\forall int j; right < j && j < a.length; a
          ↘ [i] <= a[j]));
      @ assignable less, great, pivot, a[left..right];
      @ measured_by right - left + 1;
      @*/
    static void sort(int[] a, int left, int right) {
        if (left < right) {
            split(a, left, right);
            int lless = less;
            int lgreat = great;
```

```
        int lpivot = pivot;

        if(left < less-1) sort(a, left, lless-1);

        if(lgreat+1 < right) sort(a, lgreat+1, right);
    }
}

/*@
  @ normal_behaviour
  @ requires 0 <= left && left < right && right < a.length;
  @ requires (\forall int i; 0 <= i && i < left; (\forall int j; left <= j && j < a.length; a[
        ↘ i] <= a[j]));
  @ requires (\forall int i; 0 <= i && i <= right; (\forall int j; right < j && j < a.length;
        ↘ a[i] <= a[j]));
  @ ensures (\forall int i; left <= i && i < less; a[i] < pivot);
  @ ensures (\forall int j; less <= j && j <= great; a[j] == pivot);
  @ ensures (\forall int l; great < l && l <= right; a[l] > pivot);
  @ ensures (\forall int i; 0 <= i && i < left; (\forall int j; left <= j && j < a.length; a[i
        ↘ ] <= a[j]));
  @ ensures (\forall int i; 0 <= i && i <= right; (\forall int j; right < j && j < a.length; a
        ↘ [i] <= a[j]));
  @ ensures left <= great && less <= right;
  @ ensures left <= less && great <= right;
  @ ensures less <= great;
  @ assignable less, great, pivot, a[left..right];
  @*/
static void split(int[] a, int left, int right) {
    pivot = a[left];
    less = left;
    great = right;

    /*@
      @ loop_invariant
      @     less < k && k <= great + 2
      @  && 0 <= left && left < right && right < a.length
      @  && left <= less && great <= right
      @  && left <= great && less <= right
      @  && less <= great
      @  && (\forall int i; left <= i && i < less; a[i] < pivot)
      @  && (\forall int j; less <= j && j < k && j <= great; a[j] == pivot)
      @  && (\forall int l; great < l && l <= right; a[l] > pivot)
      @  && (\forall int i; 0 <= i && i < left; (\forall int j; left <= j && j < a.length; a[i
           ↘ ] <= a[j]))
      @  && (\forall int i; 0 <= i && i <= right; (\forall int j; right < j && j < a.length; a
           ↘ [i] <= a[j]));
      @ assignable a[left..right], less, great;
      @ decreases great + 100 - k;
      @*/
    for (int k = less + 1; k <= great; ++k) {   //changed init to k = less+1 because first
          ↘ step was always omitted anyway
        if (a[k] == pivot) {
            continue;
        }
        int ak = a[k];
        if (ak < pivot) { // Move a[k] to left part
            a[k] = a[less];
            a[less] = ak;
            ++less;
        } else { // a[k] > pivot - Move a[k] to right part
            case_right(a, left, right, k, pivot);
        }
        //System.out.println("left: " + left + " less: " + less + " k: " + k + " great: " +
              ↘ great + " right: " + right);
    }
}

/*@
  @ normal_behaviour
  @ requires 0 <= left && left < right && right < a.length;
  @ requires less < k && k <= great
  @ && left <= less && left <= great &&  great <= right
  @ && (\forall int i; left <= i && i < less; a[i] < pivot)
  @ && (\forall int j; less <= j && j < k; a[j] == pivot)
  @ && (\forall int l; great < l && l <= right; a[l] > pivot);
  @ requires a[k] > pivot; //path condition
  @ requires k <= great; //loop condition
  @ requires (\forall int i; 0 <= i && i < left; (\forall int j; left <= j && j < a.length; a[
        ↘ i] <= a[j]));
```

```
      @ requires (\forall int i; 0 <= i && i <= right; (\forall int j; right < j && j < a.length;
          ↘ a[i] <= a[j]));
      @ ensures less <= k && k <= great + 1
      @ && left <= less && great <= right
      @ && (\forall int i; left <= i && i < less; a[i] < pivot)
      @ && (\forall int j; less <= j && j < k; a[j] == pivot)
      @ && (\forall int l; great < l && l <= right; a[l] > pivot);
      @ ensures great < \old(great);
      @ ensures \old(less) <= less;
      @ ensures a[k] == pivot || great < k;
      @ ensures left <= great && less <= right;
      @ ensures (\forall int i; 0 <= i && i < left; (\forall int j; left <= j && j < a.length; a[i
          ↘ ] <= a[j]));
      @ ensures (\forall int i; 0 <= i && i <= right; (\forall int j; right < j && j < a.length; a
          ↘ [i] <= a[j]));
      @ ensures less <= great;
      @ assignable great, less, a[less..great];
      @*/
    static void case_right(int[] a, int left, int right, int k, int pivot) {
        int ak = a[k];
        /*@
          @ loop_invariant
          @     a[k] > pivot
          @  && a[k-1] == pivot
          @  && 0 <= left && right < a.length
          @  && left <= less && great <= right
          @  && left <= great
          @  && less < k && k <= great+1
          @  && (\forall int i; great < i && i <= right; a[i] > pivot)
          @  && (\forall int i; 0 <= i && i < left; (\forall int j; left <= j && j < a.length; a[i
              ↘ ] <= a[j]))
          @  && (\forall int i; 0 <= i && i <= right; (\forall int j; right < j && j < a.length; a
              ↘ [i] <= a[j]));
          @ assignable great;
          @ decreases great;
          @*/
        while (a[great] > pivot) {
            --great;
        }  // if great < k here, split() is done. Changed the code to do that properly
        if (great < k) {}
        else if (a[great] < pivot) { // a[great] <= pivot
            a[k] = a[less];
            a[less] = a[great];
            ++less;
            a[great] = ak;
            --great;
        } else { // a[great] == pivot
            a[k] = pivot;
            a[great] = ak;
            --great;
        }
    }
}
```

A.3 SwapValues.java

This class contains the algorithm, occurring within the dual pivot partition, which swaps all elements equal to the pivot elements to the sides of the array range to be sorted. This is done when the central part resulting from the dual pivot partition is larger than $\frac{5}{7}$ of the array range.

```
class SwapValues {

    static int less, great, pivot1, pivot2;
    /*@
      @ normal_behaviour
      @ requires 0 < l && l < g && g < a.length-1;
      @ requires p1 < p2;
      @ requires (\forall int i; l <= i && i <= g; p1 <= a[i] && a[i] <= p2);
      @ requires (\exists int j; l <= j && j <= g; a[j] == p1);
      @ requires (\exists int m; l <= m && m <= g; a[m] == p2);
      @ ensures (\forall int i; l <= i && i < less; a[i] == pivot1);
      @ ensures (\forall int j; less <= j && j <= great; pivot1 < a[j] && a[j] < pivot2);
      @ ensures (\forall int m; great < m && m < g; a[m] == pivot2);
```

```
  @ assignable less, great, pivot1, pivot2, a[l..g];
  @*/
static void swapValues(int[] a, int l, int g, int p1, int p2) {
    // for easier specification
    less = l;
    great = g;
    pivot1 = p1;
    pivot2 = p2;

    move_less_right(a,l,g);
    move_great_left(a,l,g,less);    //last parameter is meaningless here

    outer:

    /*@
      @ loop_invariant
      @     0 < less && l <= less && 0 <= great && great <= g && less <= g && pivot1 < pivot2
      @  && (\exists int m; l <= m && m <= g; a[m] == pivot2)
      @  && less-1 <= k && k <= great+1 && k < a.length
      @  && (\forall int i; l <= i && i < less; a[i] == pivot1)
      @  && (\forall int j; less <= j && j < k+1; pivot1 < a[j] && a[j] < pivot2)
      @  && (\forall int t; k+1 <= t && t <= great; pivot1 <= a[t] && a[t] <= pivot2)
      @  && (\forall int m; great < m && m <= g; a[m] == pivot2);
      @ decreases great - k;
      @ assignable less, great, a[l..g];
      @*/
    for (int k = less - 1; ++k <= great; ) {
        int ak = a[k];
        if (ak == pivot1) { // Move a[k] to left part
            case_left(a,k,l,g);
        } else if (ak == pivot2) { // Move a[k] to right part
            move_great_left(a,l,g,k);
            if (great == k) {
                great--;
                break outer; // We're done here
            }
            case_right(a,k,l,g);
        }

    }
}


static void case_right(int[] a, int k, int l, int g) {
    int ak = a[k];
    if (a[great] == pivot1) { // a[great] < pivot2
        a[k] = a[less];
        a[less] = pivot1;
        ++less;
    } else { // pivot1 < a[great] < pivot2
        a[k] = a[great];
    }
    a[great] = ak;
    --great;
}

static void case_left(int[] a, int k, int l, int g) {
    int ak = a[k];
    a[k] = a[less];
    a[less] = ak;
    ++less;
}

/*@
  @ normal_behaviour
  @ requires 0 <= less && less < great && great <= g && great < a.length;
  @ requires pivot1 < pivot2;
  @ requires (\forall int i; less <= i && i <= great; pivot1 <= a[i] && a[i] <= pivot2);
  @ requires (\exists int j; less <= j && j <= great; a[j] == pivot1);
  @ requires (\exists int m; less <= m && m <= great; a[m] == pivot2);
  @ ensures 0 <= less && less <= great && great < a.length;
  @ ensures (\forall int i; \old(less) <= i && i < less; a[i] == pivot1);
  @ ensures (\exists int m; less <= m && m <= great; a[m] == pivot2);
  @ ensures \old(less) <= less;
  @ assignable less;
  @*/
static void move_less_right(int[] a, int l, int g) {
    /*@
      @ loop_invariant
```

```
          @     0 <= less && less <= great && great < a.length
          @  && (\forall int i; \old(less) <= i && i < less; a[i] == pivot1)
          @  && (\exists int j; less <= j && j <= great; a[j] == pivot2)
          @  && \old(less) <= less;
          @ assignable less;
          @ decreases great - less;
          @*/
        while (a[less] == pivot1) {
            ++less;
        }
    }

    /*@ normal_behaviour
      @ requires great < a.length;
      @ requires k <= great;
      @ requires k >= 0;
      @ ensures  great < a.length;
      @ ensures (\forall int i; great < i && i <= \old(great); a[i] == pivot2);
      @ ensures great == k || a[great] !=  pivot2;
      @ ensures k <= great && great <= \old(great);
      @ assignable great;
      @*/
    static void move_great_left(int[] a, int l, int g, int k) {
        /*@
          @ loop_invariant
          @     great < a.length
          @  && (\forall int i; great < i && i <= \old(great); a[i] == pivot2)
          @  && k <= great && great <= \old(great);
          @ assignable great;
          @ decreases great + 1;
          @*/
        while (a[great] == pivot2 && great != k) {
            great--;
        }
    }
}
```