Quicksort Revisited

Verifying Alternative Versions of Quicksort

Razvan Certezeanu[1], Sophia Drossopoulou[1], Benjamin Egelund-Muller[1], K. Rustan M. Leino[1,2], Sinduran Sivarajan[1], and Mark Wheelhouse[1]

rc2514@imperial.ac.uk, scd@imperial.ac.uk, be514@imperial.ac.uk, leino@microsoft.com, ss7213@imperial.ac.uk, mjw03@imperial.ac.uk

[1]Imperial College London
Department of Computing

[2]Microsoft Research
Redmond

Abstract. We verify the correctness of a recursive version of Tony Hoare's `quicksort` algorithm using the Hoare-logic based verification tool Dafny. We then develop a non-standard, iterative version which is based on a stack of pivot-locations rather than the standard stack of ranges. We outline an incomplete Dafny proof for the latter.

Keywords: automated verification, algorithms, quicksort, program transformation

1 Introduction

In 1959, while working on a project for automated translation from Russian to English, Tony Hoare found a recurring need to be able to sort word sequences into alphabetical order. To tackle this problem he invented an algorithm that was significantly faster than existing alternatives. The publication of this algorithm in 1961 as "Quicksort" [7] revolutionised the way we sort, and more generally, the way we think about and develop algorithms.

Since then, quicksort has inspired practitioners and researchers alike, including the recipient of this Festschrift. The algorithm has been modified and implemented millions of times by experienced programmers and students alike in several programming languages, and has even been choreographed as a Hungarian dance [16]. As well as the fascination for its elegant and succinct presentation, it is also interesting because it involves two inner recursive calls, and thus reasoning and program transformations applied to the algorithm are non-trivial.

In 1971, Foley and Hoare presented a hand-proof of the correctness of quicksort [5], and several proofs have been developed since. Proofs for the recursive as well as the iterative setting have also been proposed by de Boer and his co-authors in [1]. Recently, in his Turing Award lecture, Lamport showed an abstract derivation of iterative quicksort [9]. More recently, and rather surprisingly, de Gouw et al. discovered a subtle bug in `Timsort`, a sorting algorithm proposed in 2002, and which is the implementation of `java.util.Arrays.sort` [13] for non-primitive types, and part of the Android platform. They discovered the bug while trying to prove the correctness of `Timsort` using the Hoare-logic based tool Key [2].

In this paper, we reason about the correctness of two versions of quicksort: a recursive version and an iterative version. We too use a Hoare logic-based tool, namely Dafny [10].

Our recursive quicksort method deviates slightly from the standard version presented in the literature, in that we split the array into three sub-arrays, the middle one of length one, and then call the function recursively on the first and third sub-arrays.

Our iterative quicksort method is, to our knowledge, novel, in that rather than storing ranges (i.e. pairs of values) in a stack, we only store the locations of the pivots (i.e. one value), thus saving both space and time.

We have used the tool Dafny to check our implementations. To facilitate the proofs, we have defined and used lemmas in the proof of the code. We have proven some, but not all of these lemmas in Dafny.

1.1 Contributions

The key contributions of our paper are as follows:

- A proof of correctness for our variant of recursive `quicksort` in Dafny.
- A new, iterative version of `quicksort` based on the pivot locations.
- A proof outline for the correctness of our iterative `quicksort` in Dafny.

The complete Dafny code for our work can be found at [3]. To the best of our knowledge, there is no existing proof of imperative recursive quicksort in Dafny before our work. However, Leino has recently developed a proof in Dafny of the standard functional recursive algorithm, as well as an alternative version of the iterative algorithm based on ranges. Both can be found in the Dafny test suite [11]. Also, to the best our knowledge, there is no exiting version of iterative quicksort based on pivots. A comparison of its efficiency with other algorithms is future work.

The rest of this paper is organized as follows:

- Section 2 presents the notation and lemmas we will be using to specify and prove `quicksort`.
- Section 3 shows three recursive versions quicksort:
  1) Recursive `quicksort` as proposed in Hoare's original paper.
  2) Recursive `quicksort` as commonly seen in the literature.
  3) Recursive `quicksort` with the variation that the two sub-ranges are off by one, and an outline of its proof of correctness.
- Section 4 shows two iterative versions of `quicksort`:
  1) Iterative `quicksort` with a stack simulation of recursion.
  2) Novel iterative `quicksort` based on a stack of pivot locations, and outline of its proof of correctness.
- Section 5 concludes the paper with an evaluation of our work and an identification of future directions of research.

2 Specifying Quicksort

We now turn to one of the most important parts of automated program verification: specifying the program we wish to implement.

2.1 Sorting – The Task

Let's start by defining the task of sorting the contents of an array.

> Given an array `a` of integers[1] we want to rearrange the array so that the elements of the array are arranged in ascending order. Additionally, we must ensure that no elements are added to or removed from the array.

2.2 Notation, Predicates and Lemmas

Throughout this paper we adopt the Dafny convention of treating arrays as pointers to sequences of values. That is, we think of the array a as a pointer to the sequence $a\,\texttt{[}0\texttt{]}, a\,\texttt{[}1\texttt{]}, a\,\texttt{[}2\texttt{]}, ..., a\,\texttt{[}|a|-1\texttt{]}$, where $|a|$ is the length of array a.

More formally, we define a notation for describing a range. For integers i, m and n:

$$i \in [m..n) \;\equiv_{def}\; m \leq i < n$$

This notation then has a natural lifting to sequences. For a sequence a, value v and integers m and n:

$$v \in a[m..n) \;\equiv_{def}\; \exists i \in [m..n).\, [\, 0 \leq i < |a| \,\wedge\, a\,\texttt{[}i\texttt{]} = v \,]$$

where $a\,\texttt{[}i\texttt{]}$ is the i^{th} value of the sequence. Note above that the range $m..n$ is capped by the length of the sequence to ensure that no invalid dereferences take place. We refer to $a[m..n)$ as a *slice*. A slice is treated as a subsequence of the original sequence and can be dereferenced as follows:

$$a[m..n)\,\texttt{[}i\texttt{]} = \begin{cases} a\,\texttt{[}m+i\texttt{]} & \text{if } 0 \leq m+i < |a| \\ \text{undefined} & \text{otherwise} \end{cases}$$

This slice notation allows us to elegantly describe interesting properties about arrays and sequences, such as:

$$\begin{aligned} a[m..n) \leq x \;&\equiv_{def}\; \forall v \in a[m..n).\, v \leq x \\ a[m..n) \leq b[p..q) \;&\equiv_{def}\; \forall v \in a[m..n).\, \forall v' \in b[p..q).\, v \leq v' \end{aligned}$$

For ease of notation, we introduce the short-hands $a[..)$, $a[..m)$, $a[m..)$ which describe a complete sequence, a sequence up to m and a sequence from m onwards, respectively. That is:

$$a[..) \;\equiv_{def}\; a[0..|a|) \qquad a[..m) \;\equiv_{def}\; a[0..m) \qquad a[m..) \;\equiv_{def}\; a[m..|a|)$$

[1] The sorting task can actually be defined for an array of any type that has a less-then-or-equal relation $\leq$.

We further adopt the notation that whenever an array reference a occurs in a context expecting a slice, it should be interpreted as the slice $a[..)$.

Note that Dafny represents sequences with the syntax $a[m..n]$, which is equivalent to the meaning of $a[m..n)$ from our notation. Therefore, whenever the terms `a[m..n]` or `a[..]` appear in our Dafny code, their meaning should be interpreted as $a[m..n)$, or $a[..)$, respectively.

We introduce a notion of *deep equality* on sequences, denoted $\approx$. This describes when two sequences have exactly the same contents. That is:

$$a[..) \approx b[..) \;\; \equiv_{def} |a| = |b| \;\wedge\; \forall i \in [0..|\mathtt{a}|).\, \mathtt{a}[i] = \mathtt{b}[i]$$

We define the concatenation of two sequences $a \mathbin{++} b$ such that:

$$|a \mathbin{++} b| = |a| + |b|$$
$$(a \mathbin{++} b)\,\mathtt{[}i\mathtt{]} = \begin{cases} a\,\mathtt{[}i\mathtt{]} & \text{if } 0 \le i < |a| \\ b\,\mathtt{[}i - |a|\mathtt{]} & \text{if } |a| \le i < |a| + |b| \\ \text{undefined} & \text{otherwise} \end{cases}$$

We define a predicate that describes when a sequence is sorted. For a sequence a and natural numbers i and j:

$$\mathit{Sorted}(\, a[i..j)\,) \;\; \equiv_{def} \forall m, n \in [0..|a|).\, [\, i \le m \le n < j \;\;\longrightarrow\;\; a\,\mathtt{[}m\mathtt{]} \le a\,\mathtt{[}n\mathtt{]}\,]$$

We also define some other useful predicates over sequences and slices. For sequences a and b, integers i, j, m and n and an arbitrary value v:

$$\begin{aligned}
\mathit{Count}(\, a[i..j),\, v\,) \;\; &\equiv_{def} |\{k \mid k \in [i..j) \wedge a\,\mathtt{[}k\mathtt{]} = v\}| \\
a[i..j) \sim b[m..n) \;\; &\equiv_{def} \forall x.\, \mathit{Count}(\, a[i..j),\, x\,) = \mathit{Count}(\, b[m..n),\, x\,) \\
\mathit{Swapped}(a[..),\, b[..),\, i,\, j) \;\; &\equiv_{def} |a| = |b| \;\wedge\; i, j \in [0..|a|) \\
&\qquad \wedge\; b\,\mathtt{[}i\mathtt{]} = a\,\mathtt{[}j\mathtt{]} \;\wedge\; b\,\mathtt{[}j\mathtt{]} = a\,\mathtt{[}i\mathtt{]} \\
&\qquad \wedge\; \forall k \in [0..|a|) \backslash \{i, j\}.\, a\,\mathtt{[}k\mathtt{]} = b\,\mathtt{[}k\mathtt{]}
\end{aligned}$$

In the above:

- $\mathit{Count}(\, a[i..j),\, v\,)$ tracks the number of times that v occurs in the slice $a[i..j)$.
- $a[i..j) \sim b[m..n)$ states that slice $a[i..j)$ is a *permutation* of slice $b[m..n)$.
- $\mathit{Swapped}(a[..),\, b[..),\, i,\, j)$ states that the sequences $a[..)$ and $b[..)$ are exactly the same except that the elements at positions i and j have been *swapped*.

All the operators and predicates above are available, or can be easily encoded, in Dafny. However, they cannot always be written in infix or symbolic notation.

Finally, we present some useful properties of sequences and their related predicates. The following hold for all sequences a and b and for all integers i, j, k, l, m and n:

Deep Equality:

$$a \approx b \;\longrightarrow\; b \approx a \qquad\qquad a \approx b \wedge b \approx c \;\longrightarrow\; a \approx c$$
$$a \approx b \;\longrightarrow\; |a| = |b| \qquad\qquad a \approx b \;\longrightarrow\; a \sim b$$

Ranges:

$$a \approx b[0..i){+\!+}a[i..j){+\!+}b[j..|b|) \;\wedge\; m{\le}i{\le}j{\le}n \;\;\longrightarrow\;\; a \approx b[0..m){+\!+}a[m..n){+\!+}b[n..|b|)$$
$$a \approx b[0..i){+\!+}a[i..j){+\!+}b[j..|b|) \;\wedge\; a[i..j) \sim b[i..j) \;\;\longrightarrow\;\; a \sim b$$
$$a \approx a[0..i){+\!+}b[i..j){+\!+}a[j..|a|) \;\wedge\; b \approx c \;\;\longrightarrow\;\; a \approx a[0..i){+\!+}c[i..j){+\!+}a[j..|a|)$$

Permutation:

$$a \sim b \;\longrightarrow\; b \sim a$$
$$a \sim b \wedge b \sim c \;\longrightarrow\; a \sim c$$
$$a \sim b \;\longrightarrow\; |a| = |b|$$

Swapping:

$$\mathit{Swapped}(a,\, b,\, i,\, i) \;\longrightarrow\; a \approx b$$
$$\mathit{Swapped}(a,\, b,\, i,\, j) \;\longrightarrow\; a \sim b$$

Sorting:

$$\mathit{Sorted}(\, a[i..j)\,) \wedge i \le m \wedge n \le j \;\longrightarrow\; \mathit{Sorted}(\, a[m..n)\,)$$

2.3 Specifying Methods

Method specifications consist of a *Precondition*, expected to hold before the method is executed, and a *Postcondition*, that the code must ensure holds after the method terminates. We use the Dafny keywords `requires` and `ensures` to refer to the precondition and postcondition of a method respectively. We use the Dafny keyword `assert` within our code to introduce assertions, or *mid-conditions*. We also use the Dafny keywords `decreases` and `invariant` to introduce *variants* and *invariants* for loops and recursive methods.

Given some code $\mathbb{C}$ with precondition P and postcondition Q, we adopt the *total correctness* interpretation of such a specification[12], whereby

> For all program states that satisfy the precondition P, the code $\mathbb{C}$ will run without faulting and will terminate in a program state that satisfies the postcondition Q.

Sometimes, in our specifications, we need to refer to both the current and initial values of some variables. For example, in the code snippet `x := x+3`, the new value of `x` depends on its previous value. By default, all of our specifications refer to the current values of variables. As in Dafny, we use the keyword $\mathit{old}(\,.\,)$ to indicate the value before a method call. For example, $\mathit{old}($ `x` $)$ represents the value of the program variable `x` before the call to the current method. Notice that arrays are pointers to sequences. So, if we have an array a, the term $\mathit{old}($ `a` $)$ is the value of the pointer before the call, $\mathit{old}($ `a` $)[..]$ represents the current contents of the pointer before the call, while $\mathit{old}($ `a`$[..]$ $)$ represents the contents of the array before the call.

When writing specifications we use both Dafny syntax and normal mathematical notation as well as our sequence notation as developed in section 2.2. For example, we write $\forall$ and $\wedge$ rather than `forall` and `&&`.

2.4 The Specification

Sorting is specified as follows

```
method quicksort(a:array<int>)
requires a ≠ null ∧ |a| > 0
ensures a[..) ∼ old(a[..)) ∧ Sorted(a[..))
```

This specification requires that the input be a non-null, non-empty array (to rule out pathological input) and ensures that the resulting array is sorted. Additionally, the specification states that no elements are added to or deleted from the array.

3 Recursive Quicksort

Having identified the task that we need to solve, we now provide several different implementations of quicksort, ranging from classic to more inventive solutions.

The fundamental idea behind the quicksort algorithm is "divide-and-conquer":

1. Choose an element in the list – this element serves as the pivot. Set it aside (e.g. move it to the beginning or end).
2. Partition the array of elements into two sets – those less than the pivot and those greater than or equal to the pivot
3. Repeat steps 1 and 2 on each of the two resulting partitions until each set has one or fewer elements.

3.1 The Original Quicksort

Hoare's original quicksort program, as published in [7], is given as:

```
method quicksort(a:array<int>, from:nat, to:nat) {
  if (from < to) then {
    var i,j := partition(a, from, to);
    quicksort(a, from, i);
    quicksort(a, j, to);
  }
}
```

To sort the whole array, `from` should be set to 0 and `to` should be set to |`a`|.

The code presented above makes use of a variant `partition` method that does not require the caller to provide a pivot value and returns a pair rather than a single value. The pivot value is selected arbitrarily from the range [`from`..`to`). The returned pair specifies a range [`i`..`j`) of values that are equal to the chosen pivot, with elements in the range [`from`..`i`) less than the chosen pivot and [`j`..`to`) greater than the chosen pivot. More formally this can be specified as:

$$\mathtt{a}[\mathtt{m}..\mathtt{i}) < \mathtt{a}[\mathtt{i}..\mathtt{j}) < \mathtt{a}[\mathtt{j}..\mathtt{n}) \wedge \exists v.\, [\, \mathtt{a}[\mathtt{i}..\mathtt{j}) = v\,]$$

The standard version has also been studied in [1]. More recently, the original version is not seen that often. This is perhaps due to the fact that when the array has no duplicate elements, then the middle range will have length 1. i.e. $\mathtt{j} = \mathtt{i} + 1$. The algorithm then behaves like the “standard” `quicksort`, which we discuss next.

3.2 The Standard Quicksort

Usually [15], `quicksort` is presented with a method wrapper and uses a variant of `partition` which requires a pivot and returns only one value.

```
method quicksort(a:array<int>){
  quicksort(a, 0, a.Length)
}

method quicksort(a:array<int>, from:nat, to:nat) {
  if (from < to) then {
    var mid:int := partition(a, from, to, a[from]);
    quicksort(a, from, mid);
    quicksort(a, mid, to);
  }
}
```

Partition: The `partition` method rearranges an array within set bounds according to a pivot value, whilst leaving the rest of the array unmodified. This rearrangement places all elements that are smaller than the pivot before all elements that are greater than or equal to the pivot. The method returns the array index of the first element in the slice which is greater than or equal to the pivot. It is specified as follows:

```
method partition(a:array<int>, from:nat, to:nat, pivot:int)
                                                    returns (r:nat)
requires a ≠ null ∧ 0 ≤ from ≤ to ≤ |a|
ensures from ≤ r ≤ to
          ∧ a[from..r) < pivot ≤ a[r..to) ∧  a[..) ~ old( a[..) )
          ∧ a ≈ old( a[0..from) ) ++ a[from..to) ++ old( a[to..|a|) )
```

Note that in the case where all elements in the range are smaller than the pivot, the method will return $\mathtt{r} = \mathtt{to}$. Similarly, when all elements in the range are greater than or equal to the pivot, the method will return $\mathtt{r} = \mathtt{from}$.

3.3 Quicksort – Our Version

Below we show our version of recursive `quicksort`. In fact, this version was shown to us by Krysia Broda. It is very similar to the standard version, but with a little twist added: our version splits the array into three, rather than two parts: one part that is smaller than, one part that is equal to, and one part that is greater than or equal to, the pivot. Then, the recursive calls need only be called on the first and the third sub-part; the pivot remains where it was placed by `swap` in the current iteration.

Swap: The `swap` method switches the places of two elements within an array, while leaving the rest of the array unmodified. It is specified, making use of our *Swapped* predicate defined in section 2.2, as follows:

```
method swap(a:array<int>, i:nat, j:nat)
requires a ≠ null ∧ i, j ∈ [0..|a|)
ensures Swapped( a[..), old( a[..) ), i, j)
```

The Code: In the listing below we give assertions about the state of the variables at the corresponding program points, shown in green. The full Dafny code for the example below, together with the definitions of all the predicates used can be found at [3].

```
method quicksort(a:array<int>, from:nat, to:nat)
requires a ≠ null ∧ 0 ≤ from ≤ to ≤ |a|
modifies a
ensures a ≈ old( a[0..from) ) ++ a[from..to) ++ old( a[to..|a|) )
          ∧ a[..) ∼ old( a[..) ) ∧ Sorted( a[from..to) )
decreases to − from
{
  var a₀:seq<int> := a[..];
  if (from + 1 < to) {
    var pivot:int := a[from];
    assert a ≈ a₀ ∧ pivot = a[from] ∧ a[..) ∼ old( a[..) )

    var mid:int := partition(a, from + 1, to, pivot);
    assert from + 1 ≤ mid ≤ to ∧ pivot = a[from]
            ∧ a[from + 1..mid) < pivot ≤ a[mid..to)
            ∧ a ≈ a₀[0..from + 1) ++ a[from + 1..to) ++ a[to..|a|)
            ∧ a[..) ∼ old( a[..) )

    swap(a, from, mid - 1);
    assert from ≤ mid − 1 ≤ to
            ∧ a[from..mid − 1) < a[mid-1] ≤ a[mid..to)
            ∧ a ≈ a₀[0..from) ++ a[from..to) ++ a₀[to..|a|)
            ∧ a[..) ∼ old( a[..) )

    quicksort(a, from, mid - 1);
    assert from ≤ mid − 1 ≤ to
            ∧ a[from..mid − 1) < a[mid-1] ≤ a[mid..to)
28              ∧ a ≈ a0[0..from) ++ a[from..to) ++ a0[to..|a|)
29              ∧  a[..) ∼ old( a[..) ) ∧ Sorted( a[from..mid − 1) )
30
31      quicksort(a, mid, to);
32      assert a ≈ a0[0..from) ++ a[from..to) ++ a0[to..|a|)
33              ∧  a[..) ∼ old( a[..) ) ∧ Sorted( a[from..to) )
34    }
35  }
```

In **Fig. 1.** we show the assertions at several program points diagrammatically:

- PRE: before the method call (i.e. the precondition)
- MID_2: after the call of `partition` (i.e. at line 14)
- MID_3: after the call of `swap` (i.e. at line 20)
- MID_4: after the first recursive call of `quicksort` (i.e. at line 26)
- MID_5: after the second recursive call of `quicksort` (i.e. at line 32)
- POST: as an implication of the previous assertion (i.e. again at line 32)

We use `F`, `T` for `from` and `to`, and `K` as a shorthand for `mid-1`.

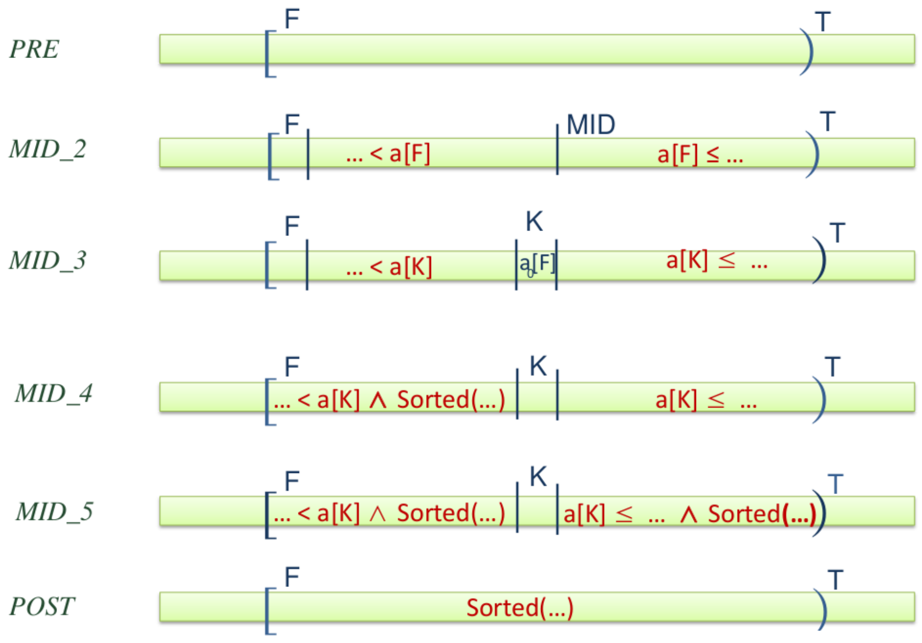

Fig. 1. Diagrammatic assertions for our recursive quicksort program.

Verification: We have verified the above code using Dafny. In order to do this, we defined and used four lemmas. We show below how the verification works: we have included in `green` the definition of auxiliary variables (lines 8, 12, 17, 21 and 25 below), and the calls of the lemmas (lines 18, 22, 26 and 27 below). The complete Dafny code can be found at [3].

```
method quicksort(a:array<int>, from:nat, to:nat)
 requires $a \neq$ null $\wedge\ 0 \leq$ from $\leq$ to $\leq$ |a|
 modifies a
 ensures a $\approx$ $old($ a[0..from) $)$ ++ a[from..to) ++ $old($ a[to..|a|) $)$
          $\wedge$ a[..) $\sim$ $old($ a[..) $) \wedge Sorted($ a[from..to) $)$
 decreases to $-$ from
 {
   var a$_0$:seq<int> := a[..];
   if (from + 1 < to) {
     var pivot:int := a[from];
     var mid:int := partition(a, from + 1, to, pivot);
     var a$_1$:seq<int> := a[..];

     swap(a, from, mid - 1);
     var a$_2$:seq<int> := a[..];
     L_swap_impl_sameUpTo(a$_2$, a$_1$, from, mid -1);

     quicksort(a, from, mid - 1);
     var a$_3$:seq<int> := a[..];
     L_sameUpTo_prsrv_less(a$_3$, a$_2$, pivot, mid, to);

     quicksort(a, mid, to);
     var a$_4$:seq<int> := a[..];
     L_sameUpTo_prsrv_grEq(a$_4$, a$_3$, pivot, mid, to);
     L_conc_impl_Sorted(a$_4$, from, mid, to);
   }
 }
```

From the eighteen assertions mentioned in the code, Dafny only needed help with the proofs of four, and needed no help at all for the case where `from`$+1 \geq$ `to`. We now list the lemmas used above, using the convention that a, b, c stand for sequences of type T, while $elem \in T$ is a possible value, and i, j, k, l, m and n are natural numbers.

L_swap_impl_sameUpTo(a, b, i, j,)**:**

$$|a| = |b| \wedge i \leq j < |a| \wedge a[..) \sim b[..) \wedge Swapped(a, b, i, j)$$
$$\longrightarrow \quad a \approx b[0..i) \text{ ++ } a[i..j+1) \text{ ++ } b[j+1..) \wedge a[..) \sim b[..)$$

This lemma says that swapping creates a permutation of the original array, leaving the $[..i)$ and the $[i+1..)$ range unmodified. The proof follows by unfolding the definitions.

L_sameUpTo_prsrv_less(a, b, $elem$, m, n)**:**

$$|a| = |b| \wedge a \approx b[..m) \text{ ++ } a[m..n) \text{ ++ } b[n..) \wedge a[..) \sim b[..)$$
$$\wedge\ b[m..n) < elem$$
$$\longrightarrow \quad a[m..n) < elem$$

This lemma says that if an array a is a permutation of an array b, and is identical with b in the ranges $[..m)$ and $[n..)$, then b is smaller than $elem$ in the range $[m..n)$, then a is also smaller than $elem$ in the range $[m..n)$. The proof follows by establishing that $a[m..m) \sim b[m..m)$.

L_sameUpTo_prsrv_grEq(a, b, $elem$, m, n):

$$\begin{array}{l} |a| = |b| \;\wedge\; a \approx b[..m) +\!\!+\, a[m..n) +\!\!+\, b[n..) \;\wedge\; a[..) \sim b[..) \\ \wedge\; elem \leq b[m..n) \\ \qquad\qquad \longrightarrow\;\; elem \leq a[m..n) \end{array}$$

This lemma says that if an array a is a permutation of an array b, and is identical with b in the ranges $[..m)$ and $[n..)$, then b is greater or equal to $elem$ in the range $[m..n)$, then a is also greater or equal to $elem$ in the range $[m..n)$. The proof follows by establishing that $a[m..m) \sim b[m..m)$.

L_conc_impl_Sorted(a, i, j, k):

$$\begin{array}{l} i < j \leq k \leq |a| \;\wedge\; i < |a| \;\wedge\; Sorted(\,a[i..j-1)\,) \;\wedge\; Sorted(\,a[j..k)\,) \\ \wedge\; a[i..j-1) < a[j-1] \leq a[j..) \\ \qquad\qquad \longrightarrow\;\; Sorted(\,a[i..k)\,) \end{array}$$

This lemma says that concatenation of two sorted sub-ranges $[i..j-1)$ and $[j..k)$, where the left sub-range contains smaller elements than the element at $a[j-1]$, and where $a[j-1]$ is smaller or equal to the elements at $[j..k)$ produces a sorted range $[i..k)$. The proof follows by unfolding the definitions.

4 Iterative Quicksort

An iterative version of quicksort can be obtained from the recursive one directly by applying the standard transformation of recursion. This is shown in section 4.1. A more interesting (and more efficient) iterative version can be obtained if we observe some properties of the first version. This is shown in section 4.2.

4.1 Iterative Quicksort Version 1 – Simulating Method Arguments

The Code: We use a stack, here called `memos`, to keep track of the parameters of the recursive method. We simulate the push/pop operations by decrementing/incrementing the value of `top`. We start by pushing `0` and |`a`| onto `memos` (lines 9 and 10). Then, we read the values of `from` and `to` iteratively from the stack (lines 13 and 14), until the stack is empty. The first recursive call, `quicksort(a,f,mid-1)`, is represented by pushing the values `from` and `mid` $-$ 1 onto the stack (lines 21 and 22), and the second recursive call, `quicksort(a,mid,to)`, is represented by pushing the values `to` and `mid` onto the stack (lines 23 and 24).

```
method quicksort(a:array<int>)
requires a ≠ null ∧ |a| > 0
modifies a
ensures  a[..) ~ old( a[..) ) ∧ Sorted( a[..) )
{
  var len:int := 2 * a.Length
  var memos:array<int> := new int[len];
  var top:int := len - 2;
  memos[top] := 0;
  memos[top + 1] := a.Length;

  while( top < len ){
    var from:int := memos[top];
    var to:int := memos[top + 1];
    top := top + 2;
    if (from + 1 < to) {
      var pivot:int := a[from];
      var mid:int := partition(a, from + 1, to, pivot);
      swap(a, from, mid - 1);
      top := top - 4;
      memos[top] := from;
      memos[top + 1] := mid - 1;
      memos[top + 2] := mid;
      memos[top + 3] := to;
    }
  }
}
```

We sketch the loop invariant for this version of `quicksort` in **Fig. 2.**, but do not discuss the verification in more detail.

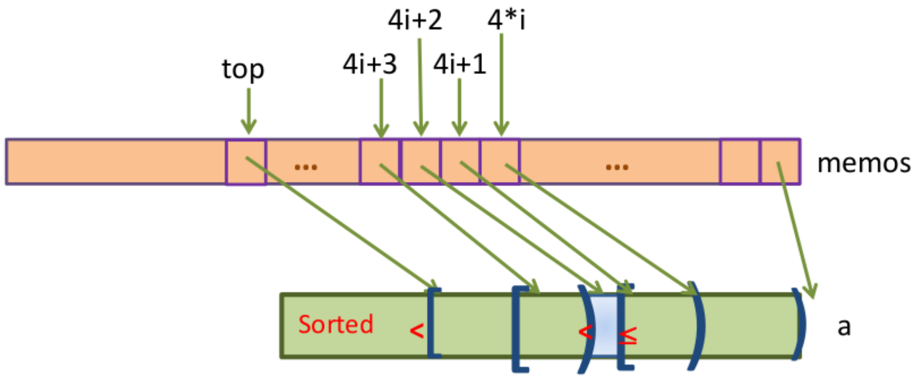

Fig. 2. Invariant sketch for our iterative "simulated recursion" `quicksort` program.

4.2 Iterative Quicksort Version 2 – Pivot Storage

Preliminaries: We now discuss the second version of iterative `quicksort`, which, to the best of our knowledge, is novel. Rather than just translating the recursion into iteration, as we did in section 4.1, we instead draw inspiration from observing the following two facts about the code from section 4.1: Firstly, neighbouring `to` and `from` values are off by 1 - this can be seen in lines 22 and 23. Secondly, after swapping the array elements at `from` and $(\mathtt{mid} - 1)$ (line 19), the contents of the array at $(\mathtt{mid} - 1)$ never changes.

This led us to the idea that, rather than pushing and popping the ranges on which we operate (i.e. the values `from` and `to`) we can instead work with the final location of the pivot $(\mathtt{mid} - 1)$. We know that the contents of the array at this location will not change, and we also know that the next range to operate on will start at the location succeeding the location of the current pivot. Therefore, we use an array of pivot locations, called `pivs`.

We know that `pivs` contains strictly increasing values:

$$\forall i, j \in [0..|\mathtt{pivs}|).\, [\, i < j \to \mathtt{pivs}[i] < \mathtt{pivs}[j]\,]$$

We also know the pivot locations delineate array segments with increasing values and that the contents of array `a` at location `pivs[i]` will not change in subsequent iterations, since all the values preceding it are smaller, and all values coming after it are greater of equal. We encode these two properties as follows:[2]

$$\forall i \in [\mathtt{top}..|\mathsf{a}|).\, \mathsf{a}[..\mathtt{pivs}[i]) < \mathtt{a}[\mathtt{pivs}[i]] \leq \mathsf{a}[\mathtt{pivs}[i] + 1..)$$

We use the variable `top` with values from the interval $[0..|\mathsf{a}| + 1)$, to range over the indices of the array `pivs`, so that the contents of the slice $\mathtt{pivs}[\mathtt{top}+1..)$ is always defined. We initialize `top` with $|\mathsf{a}|$. We increment `top` in order to pop a pivot location, and decrement it in order to push a pivot location. This gives us the invariant:

$$0 \leq \mathtt{top} \leq |\mathsf{a}|$$

We also use variables `from` and `to` to delineate the range we are currently operating on. We have the invariants that

$$0 \leq \mathtt{top} \leq |\mathsf{a}| \,\wedge\, 0 \leq \mathtt{from} \leq \mathtt{to} = \mathtt{pivs}[\mathtt{top}] \,\wedge\, \mathtt{pivs}[|\mathsf{a}|] = |\mathsf{a}|$$

that the array is sorted up to and including the index `from`, and that all values before `from` are smaller or equal to those starting `from` and onwards:

$$\mathit{Sorted}(\, \mathsf{a}[..\mathtt{from} + 1)\,) \,\wedge\, \mathsf{a}[..\mathtt{from}) \leq \mathsf{a}[\mathtt{from}..)$$

Note: while the contents of array `a` at location `pivs[i]` will not change in subsequent iterations, the contents of `a` at location `from` *might* change at subsequent iterations, as it is possible that $\mathsf{a}[\mathtt{from}] > \mathsf{a}[\mathtt{from}+k]$ for some $k \in \mathbb{N}$.

[2] The careful reader will notice that the array look-up $\mathtt{a[pivs[}i\mathtt{]+1]}$ is not always defined. Nevertheless, the assertion is well-formed, because it stands for $\forall i \in [\mathtt{top}..|\mathsf{a}|).\forall j \in [0..\mathtt{pivs}[i]).\forall k \in [\mathtt{pivs}[i]\mathtt{+1..|a|}).\, \mathtt{a}[j] < \mathtt{a}[\mathtt{pivs}[i]] \leq \mathtt{a}[k]$

The Code: The deliberations from above lead us to the code below. Essentially, we have a loop which either increases `from`, or decreases the distance between `to` and `from`. The loop terminates when $\texttt{a.Length} - \texttt{from} \leq 1$, which, given the invariants from above, implies that $\mathit{Sorted}(\,\texttt{a}[..|\texttt{a}|)\,)$. The loop invariant consists of nine conjuncts.

```
method quicksort(a:array<int>)
requires $a \neq \texttt{null} \wedge |\texttt{a}| > 0$
modifies a
ensures $\texttt{a}[..) \sim \mathit{old}(\,\texttt{a}[..)\,) \wedge \mathit{Sorted}(\,\texttt{a}[..)\,)$
{
  var pivs:array<int> := new int[a.Length+1];
  pivs[a.Length]:= a.Length;
  var from, to, top := 0, a.Length, a.Length;

  while (a.Length - from > 1)
  invariant $0 \leq \texttt{top} \leq |\texttt{a}| \wedge 0 \leq \texttt{from} \leq \texttt{to} = \texttt{pivs[top]}$
        $\wedge\ \texttt{pivs}[|\texttt{a}|] = |\texttt{a}| \ \ \wedge\ \ \forall i \in [\texttt{top}..|\texttt{a}|+1).\,\texttt{pivs}[i] \leq i+1$
        $\wedge\ \forall i,j \in [\texttt{top}..|\texttt{a}|+1).\,[\,i<j \rightarrow \texttt{pivs}[i] < \texttt{pivs}[j]\,]$
        $\wedge\ \forall i \in [\texttt{top}..|\texttt{a}|).\,\texttt{a}[..\texttt{pivs}[i]\,) < \texttt{a}[\texttt{pivs}[i]] \leq \texttt{a}[\texttt{pivs}[i]..)$
        $\wedge\ \texttt{a}[..\texttt{from}) \leq \texttt{a}[\texttt{from}..)$
        $\wedge\ \ \texttt{a}[..) \sim \mathit{old}(\,\texttt{a}[..)\,) \wedge \mathit{Sorted}(\,\texttt{a}[..\texttt{from}+1)\,)$
  decreases $|\texttt{a}| - \texttt{from},\ \texttt{to} - \texttt{from}$
  {

    if ( (to - from) <= 1 ) {
      L_sorted_combine(a, from, to);
      L_prsrv_pivot(a,to);

      from := to + 1;
      top := top + 1;
      to := pivs[top];
    } else {
      var a$_2$:seq<int> := a[..];

      var pivot:int := a[from];
      var mid:nat := partition(a, from + 1, to, pivot);
      var a$_2$:seq<int> := a[..];

      swap(a, from, mid - 1);
      var a$_3$:seq<int> := a[..];
      L_swap_prsrv_less(a$_3$, a$_2$, from, mid);
      L_sameUpTo_trans(a$_3$, a$_2$, a$_1$, from, to);
      L_sameUpTo_prsv_sorted(a$_3$, a$_1$, from, to);

      top := top - 1;
```

```
        pivs[top] := mid - 1;
        to := mid - 1;
      }
    }
  }
```

We sketch the loop invariant for this version of quicksort in **Fig. 3.**.

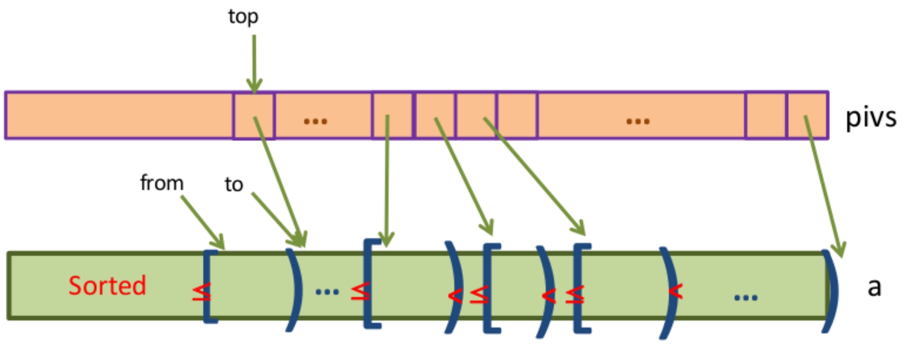

Fig. 3. Invariant sketch for our iterative "pivot storage" quicksort program.

Verification: In our Dafny proof we wrote twenty-four assert statements to guide the prover, and called five lemmas at the code locations listed above. The lemmas are given below and proven in the next subsection. In the following, a, b and c stand for sequences, while i, j, k, m and n are natural numbers.

L_sorted_combine(a, m, n)**:**

$$m \le n \le m+1 \ \wedge\ Sorted(\, a[..m+1)\,) \ \wedge\ a[..n) < a[n] \le a[n+1..) \\ \longrightarrow\ \ Sorted(\, a[..m+2)\,)$$

The lemma above increases the range for which we know that an array a is sorted.

L_prsrv_pivot(a, m)**:**

$$m < |a| \ \wedge\ a[..m) < a[m] \le a[m+1..) \quad\longrightarrow\quad a[..m+1) \le a[m+1..)$$

The lemma above increases the range for which we know that elements are smaller than the elements in the remaining array.

L_swap_prsrv_less(a, b, m, n)**:**

$$m < n \le |b| \ \wedge\ b[..m) < b[m..) \ \wedge\ b[m+1..n) < b[m] \ \wedge\ |a| = |b| \\ \wedge\ Swapped(a,\, b,\, m,\, n-1) \\ \longrightarrow\ \ a[..m) < a[m..) \ \wedge\ a[m..n-1) < a[n-1]$$

The lemma above asserts that, after swapping, a pivot correctly partitions the array. The left subsequence is smaller than the right subsequence and the middle subsequence is smaller than the element $a[n-1]$.

L_sameUpTo_trans(a, b, c, m, n):

$$\begin{aligned}
&|a| = |b| = |c| \wedge m < n \leq |a| \wedge a \approx b[..m) \mathbin{+\!+} a[m..n) \mathbin{+\!+} b[n..) \\
&\wedge\ a[..) \sim b[..) \wedge b \approx c[..m+1) \mathbin{+\!+} b[m+1..n) \mathbin{+\!+} c[n..) \\
&\wedge\ b[..) \sim c[..) \\
&\qquad\qquad \longrightarrow\ a \approx c[..m) \mathbin{+\!+} a[m..n) \mathbin{+\!+} c[n..) \wedge\ a[..) \sim c[..)
\end{aligned}$$

The lemma above asserts that permutation, and array composition from subarrays are transitive relations.

L_sameUpTo_prsv_sorted(a, b, i, j):

$$\begin{aligned}
&|a| = |b| \wedge i < j \leq |b| \wedge \mathit{Sorted}(\,b[..i+1)\,) \wedge a[..i) \leq a[i..) \\
&\wedge a \approx b[..i) \mathbin{+\!+} a[i..j) \mathbin{+\!+} b[j..) \\
&\qquad\qquad \longrightarrow\ \mathit{Sorted}(\,a[..i+1)\,)
\end{aligned}$$

This lemma ensures that swapping preserves sortedness of sub-ranges of the array.

4.3 Proofs

We now show the proofs of these lemmas.

Proof of L_sorted_combine(a, m, n):

Given

(1) $m \leq n \leq m+1$

(2) $\mathit{Sorted}(\,a[..m+1)\,)$

(3) $a[..n) < a[n] \leq a[n+1..)$

To show

(A) $\mathit{Sorted}(\,a[..m+2)\,)$

From (1), we obtain that either $m = n$ or $m+1 = n$. We proceed by case analysis.

1st Case:

(4) $m = n$

Then we have

(5) $a[..m) < a[m] \leq a[m+1..)$	from (3) and (4)
(6) $a[m] < a[m+1]$	from (5)
(A) $\mathit{Sorted}(\,a[..m+2)\,)$	from (2) and (6)

2nd Case:

(4) $m+1 = n$

Then we have

(5) $a[..m+1) < a[m+1]$	from (3) and (4)
(A) $\mathit{Sorted}(\,a[..m+2)\,)$	from (2) and (5)

Proof of L_prsrv_pivot(a, m): by unfolding the definitions.

Proof of L_swap_prsrv_less(a, b, m, n):

Given
(1) $m < n \leq |b|$
(2) $b[..m) < b[m..)$
(3) $b[m+1..n) < b[m] \leq b[n+1..)$
(4) $|a| = |b|$
(5) $Swapped(a,\ b,\ m,\ n-1)$
To Show
(A) $a[..m) < a[m..)$
(B) $a[m..n-1) < a[n-1] \leq a[n..)$

We obtain

(6) $a[..m) \approx b[..m)$	from (5)
(7) $a[m] = b[n-1]$	from (5)
(8) $a[m+1..n-1) \approx b[m+1..n-1)$	from (5)
(9) $a[n-1] = b[m]$	from (5)
(10) $a[n..) \approx b[n..)$	from (5)
(A) $a[..m) < a[m..)$	from (2), (7)-(10)
(11) $a[m..n-1) \approx b[n-1]$ ++ $b[m+1..n-1)$	from (7), (8)
(12) $a[m..n-1) < b[m]$	from (11), (2) and (3)
(13) $a[m..n-1) < a[n-1]$	from (12), (9)
(14) $a[n-1] \leq a[n..)$	from (3), (9) and (10)
(B) $a[m..n-1) < a[n-1] \leq a[n..)$	from (13) and (14)

Proof of L_sameUpTo_trans(a, b, c, m, n):

Given
(1) $|a| = |b| = |c|$
(2) $m < n \leq |a|$
(3) $a \approx b[..m)$ ++ $a[m..n)$ ++ $b[n..)$
(4) $a[..) \sim b[..)$
(5) $b \approx c[..m+1)$ ++ $b[m+1..n)$ ++ $c[n..)$
(6) $b[..) \sim [..c)$
To Show
(A) $a \approx c[..m)$ ++ $a[m..n)$ ++ $c[n..)$
(B) $a[..) \sim c[..)$
We obtain

(B) $a[..) \sim c[..)$	from (4) and (6)
(7) $b[..m) \approx c[..m)$	from (5), and by $m < m+1$
(8) $b[n..) \approx c[n..)$	from (5)
(A) $a \approx c[..m)$ ++ $a[m..n)$ ++ $c[n..)$	from (3), (7) and (8)

Proof of L_sameUpTo_prsv_sorted(a, b, i, j):

Given
- (1) $|a| = |b|$
- (2) $i < j \leq |b|$
- (3) $Sorted(\ b[..i+1)\ )$
- (4) $a[..i) \leq a[i..)$
- (5) $a \approx b[..i) +\!\!+\ a[i..k) +\!\!+\ b[k..)$

To Show
- (A) $Sorted(\ a[..i+1)\ )$

We obtain

(6) $Sorted(\ b[..i)\ )$	from (3) and because $i < i+1$
(7) $Sorted(\ a[..i)\ )$	from (5) and (6)
(8) $i > 1 \ \rightarrow\ a[i-1] \leq a[i]$	from (4)
(A) $Sorted(\ a[..i+1)\ )$	from (7) and (8)

5 Experiences, Conclusions and Future Work

Despite extensive testing and hand-written proofs, it was reassuring when Dafny confirmed the correctness of our `quicksort`. We found array-sequence infix operators to be useful in the development of both the algorithm and reasoning.

Dafny was extremely effective in helping us iron out many little, fiddly bugs at the original stages of our work. As we progressed, the process became both slow and addictive. Those of us new to Dafny were often surprised to see that Dafny/Z3 could automatically discharge proof obligations which were, in our opinion, non-trivial, while it was often unable to discharge what we considered trivial ones. This was due to our limited previous understanding of Z3.

We therefore proceeded in a somewhat experimental fashion. We inserted `assume` statements for all the proof obligations, and gradually replaced them by `assert` statements. When the verifier was unable to discharge an obligation, we wrote a lemma, whose validity we checked through hand-written proofs. As a result, the lemmas we have developed do not seem to be the most interesting or intuitive ones, and their choice might have been affected by the particular order in which we happened to require them.

The computational power needed for the proofs to go through was considerable. Therefore, we adopted little tricks to focus the tool on particular aspects of the proof. For example, we would replace part of the code with `assume false`, so that the tool would not need to check validity past this point. We also split the proof of the pivot-based iterative `quicksort` into two: First we replaced the code in the `else` branch by `assume false`. This let us prove that the initialization establishes the loop invariant and that the `then` branch of the loop preserves it. Then we wrote a function whose body consists of `assume` statements for all the loop invariants, followed by the code from the `else` branch of the loop and ending in `assert` statements for all the loop invariants. This let us prove that the `else` branch of the loop also preserves the loop invariant.

The experimental fashion for discovering useful lemmas, and the ticks to focus the tool on certain aspects are often seen in the Verification Corner videos [14].

We believe that Visual Studio should provide more automatic support for steering the proof effort and more help with interactive program and proof development.

As future work, we would like to complete the proofs of the lemmas we have used, complete the proofs of the other two versions of `quicksort`, and try and unify the arguments used in the various proofs. We would also like to run benchmarks to compare the efficiency of our pivot-based algorithm with that of other algorithms in the literature. Finally, we want to port the Dafny proofs to our tool Apollo [4], which maps Java, Haskell code and proof idioms onto Dafny.

Acknowledgments:

We thank Krysia Broda for showing us the recursive, non-standard version of `quicksort`, and the anonymous reviewers of this volume for valuable suggestions and pointers.

Razvan Certezeanu, Benjamin Egelund-Muller and Sinduran Sivarajan thank the Department of Computing at Imperial College for funding their Undergraduate Research Opportunities Programme (UROP) Placements, undertaken under Mark Wheelhouse's supervision, which they spent working on Apollo, and this paper.

Sophia Drossopoulou thanks Microsoft Research and Judith Bishop for a research gift and her very warm hospitality at Microsoft Research, and the EU project Upscale, FP7-612985, for supporting part of this work, and for the opportunity to collaborate with Frank S. de Boer, the recipient of this Festschrift.

References

1. Apt, K., Boer, F., Olderog, E.: Verification of sequential and concurrent programs. Springer, Dordrecht (2009).
2. Beckert, B., Hahnle, R., Schmitt, P.: Verification of Object-Oriented Software. The KeY Approach. Springer-Verlag Berlin Heidelberg, Berlin, Heidelberg (2006).
3. Certezeanu, R., Drossopoulou, S., Egelund-Muller, B., Sivarajan, S., Wheelhouse, M., Leino, K.: Dafny Code for Variations on Quicksort, `http://www.doc.ic.ac.uk/~mjw03/research/quicksort.html`.
4. Certezeanu, R., Drossopoulou, S., Egelund-Muller, B., Sivarajan, S., Wheelhouse, M., Leino, K.: Apollo: An interactive Program and Proof development tool for Java and Haskell, based on Dafny. `--toappear--`.
5. Foley, M., Hoare, C.: Proof of a recursive program: Quicksort. The Computer Journal. 14, 391-395 (1971).
6. Gouw, S., Rot, J., Boer, F., Bubel, R., Hahnle, R.: OpenJDK's Java.utils.Collection.sort() Is Broken: The Good, the Bad and the Worst Case. CAV (1), Lecture Notes in Computer Science. 9206, 273-289 (2015).
7. Hoare, C.: Algorithm 64: Quicksort. Communications of the ACM. 4, 321 (1961).
8. Hoare, C.: An axiomatic basis for computer programming. Communications of the ACM. 12, 576-580 (1969).
9. Lamort, L.: Thinking Above the Code, `https://www.youtube.com/watch?v=-4Yp3j_jk8Q`.

10. Leino, K.: Dafny: An Automatic Program Verifier for Functional Correctness. LNCS Springer. 6355, 348-370 (2010).
11. Leino, K.: Dafny: An Automatic Program Verifier for Functional Correctness, `http://dafny.codeplex.com`.
12. Manna, Z.: Mathematical theory of computation. McGraw-Hill, New York (1974).
13. Oracle Documentation: Arrays (Java Platform SE 7), `http://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html`.
14. The Verification Corner - Microsoft Research, `http://research.microsoft.com/en-us/projects/verificationcorner`.
15. Wikipedia: Quicksort, `https://en.wikipedia.org/wiki/Quicksort`.
16. YouTube: Quick-sort with Hungarian (Kkllmenti legnyes) folk dance, `https://www.youtube.com/watch?v=ywWBy6J5gz8`.