Chapter 19
Verification of Counting Sort and Radix Sort

Stijn de Gouw, Frank S. de Boer, Jurriaan Rot

Sorting is an important algorithmic task used in many applications. Two main aspects of sorting algorithms which have been studied extensively are complexity and correctness. [Foley and Hoare, 1971] published the first formal correctness proof of a sorting algorithm (Quicksort). While this is a handwritten proof, the development and application of (semi)-automated theorem provers has since taken a huge flight. The major sorting algorithms Insertion sort, Heapsort and Quicksort were proven correct by Filliâtre and Magaud [1999] using the proof assistant Coq. Recently, Sternagel [2013] formalized a proof of Mergesort within the interactive theorem prover Isabelle/HOL.

In this chapter we discuss the formalization of the correctness of *Counting sort* and *Radix sort* in KeY, based on the paper [de Gouw et al., 2014]. Counting sort is a sorting algorithm based on arithmetic rather than comparisons. Radix sort is tailored to sorting arrays of large numbers. It uses an auxiliary sorting algorithm, such as Counting sort, to sort the digits of the large numbers one-by-one. The correctness of Radix sort requires the *stability* of the auxiliary sorting algorithm. Stability here means that the order between different occurrences of the same number is preserved. To the best of our knowledge, stability has only been formalized in higher-order logic [Sternagel, 2013].

We provide the first mechanized correctness proof of Counting sort and Radix sort. Several industrial case studies have already been carried out in KeY [Ahrendt et al., 2012, Mostowski, 2005, 2007]. In contrast to most industrial code, which is large but relatively straightforward, Counting sort and Radix sort are two relatively small but ingenious and nonstandard algorithms with inherently complex correctness proofs.

19.1 Counting Sort and Radix Sort Implementation

Counting sort is a sorting algorithm based on addition and subtraction rather than comparisons. Here we consider a version of Counting sort that takes as input an array

a of *large numbers* in base-k and a column index m. Each large number is represented by an array of digits with the least significant digit first, thus a is a two dimensional array of nonnegative integers (i.e., $a[1][0]$ is the least significant digit of the second large number in a). It then sorts the large numbers in a based solely on the value of their $m+1$-th digit. Listing 19.1 shows the Java implementation. The worst-case time complexity is in $\mathcal{O}(n+k)$, where n is the number of elements in a[1].

```
1  public static int[][] countSort(int[][] a, int k, int m) {
2    int[] c = new int[k]; //initializes to zero
3    int[][] res = new int[a.length][];
4
5    for(int j=0; j<a.length; j++) {
6      c[a[j][m]]=c[a[j][m]]+1;
7    }
8    for(int j=1; j<k; j++) {
9      c[j]=c[j]+c[j-1];
10   }
11   for(int j=a.length-1; j>=0; j--) {
12     c[a[j][m]]=c[a[j][m]]-1;
13     res[c[a[j][m]]]=a[j];
14   }
15   return res;
16 }
```

Listing 19.1 Counting sort

Intuitively, the algorithm works as follows. After the first loop, for an arbitrary value $i \in [0:k-1]$, $c[i]$ contains the number of times that i occurs in column m of a. During the second loop, the partial sums of c are computed (i.e., $c[i] = c[0] + \ldots + c[i]$), so that $c[i]$ contains the number of elements in (column m of) a that are less than or equal to i. At this moment, for every value i occurring in a, $c[i]$ can thus be interpreted as being the index in the sorted array before which the large number with value i in column m should occur — if there are multiple such large numbers, then these should be placed to the left. Indeed in the final loop, c is used to place the elements of a in the resulting sorted array *res* by, for each element $a[i]$, first decreasing $c[a[i][m]]$ by one and then placing $a[i]$ at position $c[a[i][m]]$. Notice that equal elements are thus inserted from right to left — so by starting at the last element of a and counting down, the algorithm becomes *stable*. Thus, the order on the previous digits is preserved.

Figure 19.1 shows an example execution of the last loop of a call to method `countSort(a, 3, 2)`, where the input array a contains respectively the arrays 1 0 2, 2 1 1 and 0 2 1 (representing the numbers 201, 112 and 120), with digits in base $k = 3$; sorting is done based on column $m = 2$, which has the digits 2, 1 and 1 respectively. The far left shows the contents of *res* and *C* just before the first iteration of the last loop. In the first iteration, the number 120 (the last number of the input

[1] Note that this does not conflict with the well-known lower bound of $n\lg(n)$, since that holds for sorting algorithms based on comparing array elements.

Content of the array *res*:

```
null      null      2 1 1     2 1 1
null  ⟹  0 2 1  ⟹  0 2 1  ⟹  0 2 1
null      null      null      1 0 2
```

Content of the array *C*:

0 2 3 ⟹ 0 1 3 ⟹ 0 0 3 ⟹ 0 0 2

Figure 19.1 Iterations of the last loop in Counting sort with input arrays 1 0 2, 2 1 1 and 0 2 1.

```
1 0 2     0 2 1     1 0 2     2 1 1
0 2 1  ⟹  1 0 2  ⟹  2 1 1  ⟹  0 2 1
2 1 1     2 1 1     0 2 1     1 0 2
```

Figure 19.2 Successive iterations of the Radix sort loop, highlighting the column processed by `stableSort`.

array) is placed at its sorted position in row 2, as indicated by the highlighted value 2 of that step in the *C* array.

Radix sort sorts an array of large numbers digit-by-digit, using an auxiliary stable sorting algorithm `stableSort` to sort on each individual digit. This is reminiscent of the typical way one sorts a list of words: letter by letter. A suitable candidate for `stableSort` is the implementation of Counting sort given in Listing 19.1. An implementation of Radix sort is given in Listing 19.2. We assume that all large numbers in a have the same length (in particular, all of them have $a[0].\mathit{length}$ digits).

```
1 public static int[][] radixSort(int[][] a, int k) {
2   for(int i=0; i<a[0].length; i++) {
3     a = stableSort(a,k,i);
4   }
5   return a;
6 }
```

Listing 19.2 Radix sort

The call to `stableSort(a, k, i)` sorts a by the i-th column. The array a is sorted column by column, starting from the least significant digit at index 0 up to the most significant digit at index $a[0].\mathit{length}-1$. Notice that it is essential for Radix sort that this auxiliary algorithm is stable, so that the order induced by the earlier iterations is preserved on equal elements in the i-th column.

Figure 19.2 illustrates an example run of the algorithm, showing the contents of the array a after each loop iteration. The input array, representing the large numbers 201, 120 and 112 with digits in base $k = 3$, is shown on the far left and the sorted output is shown on the far right.

19.2 High-Level Correctness Proof

As we have seen in the previous section, the correctness of Radix sort depends on the *stability* of the auxiliary sorting algorithm. In this section, we formalize the property of being "stable" and give a high-level proof of the correctness and the stability of Counting sort. Subsequently we prove the correctness of Radix sort. All contracts and invariants are formalized in JML.

To avoid getting side-tracked by technicalities, and to simplify the presentation, we assume in the high-level proof that all arrays have positive length, and there are no "Array Index Out of Bounds Exceptions." We drop these assumptions in the mechanized proofs and show in Section 19.3 how this affects the specifications and corresponding correctness proofs.

The following JML contract specifies a generic sorting algorithm that sorts a two-dimensional array based solely on the numbers occurring in a given column *m*:

```
/*@ public normal_behavior
  @   requires
  @        k > 0 && 0 <= m && m < a[0].length
  @     && (\forall int row; 0 <= row && row < a.length;
  @            a[row].length == a[0].length)
  @     && (\forall int row; 0 <= row && row < a.length;
  @            0 <= a[row][m] && a[row][m] < k);
  @   ensures
  @        \dl_seqPerm(\dl_array2seq(\old(a)),
  @                    \dl_array2seq(\result))
  @     && (\forall int row;
  @            0 <= row && row < \result.length-1;
  @               \result[row][m] <= \result[row+1][m]);
  @*/
public static int[][] countSort(int[][] a, int k, int m);
```

Listing 19.3 Generic sorting contract

As explained in Section 8.1.2.9, the `\dl_` prefix is an escape sequence that allows referencing functions defined in JavaDL in JML specifications. The JavaDL function *array2seq* converts an array into a sequence and the JavaDL predicate *seqPerm* is true if the two sequences passed as parameters are permutations of each other. Chapter 5 has a precise definition of the predicate *seqPerm* .

The precondition, specified by the JML **`requires`** clause, states that all large numbers have the same length $a[0].length$, and furthermore that all digits in the m-th column are bounded by k. In the postcondition (**`ensures`**), the formula $seqPerm(array2seq(old(a)), array2seq(\backslash result))$ guarantees that the returned array $\backslash result$ is a permutation of the input array a. The second conjunct of the postcondition states that $\backslash result$ is sorted with respect to column m.

The above contract specifies correctness, but not stability. The contract below formalizes stability, by ensuring that if two different large numbers have the same

value for the $m+1$-th digit, then their original relative order from the input array is preserved:

```
/*@ public normal_behavior
  @   requires
  @       0 <= m && m < a[0].length
  @     && (\forall int row; 0 <= row && row < a.length;
  @           a[row].length == a[0].length);
  @   ensures
  @     (\forall int row; 0 <= row && row < \result.length-1;
  @              \result[row][m] == \result[row+1][m]
  @          ==> (\exists int i,j;
  @                  0 <= i && i < j && j < a.length;
  @                            \result[row]==\old(a[i])
  @                         && \result[row+1]==\old(a[j])));
  @*/
public static int[][] countSort(int[][] a, int k, int m);
```

Listing 19.4 Contract specifying stability

19.2.1 General Auxiliary Functions

For a human readable proof of Counting sort, and for both the specification and proof of Radix sort, it is absolutely crucial to introduce suitable abstractions. We therefore define the following auxiliary functions:

Name	Meaning
$val(b,r,d,a)$	$\sum_{i=0}^{d}(a[r][i]*b^i)$
$cntEq(x,r,a,c)$	$\lvert\{i \mid 0 \le i \le r \wedge a[i][c] = x\}\rvert$
$cntLt(x,a,c)$	$\lvert\{i \mid 0 \le i < a.length \wedge a[i][c] < x\}\rvert$
$pos(x,r,a,c)$	$cntEq(x,r,a,c) + cntLt(x,a,c)$

Intuitively, $val(b,r,d,a)$ is the large number represented in base b which is stored in row r of the array of large numbers a (and d is the index of the last digit). The function *cntEq* counts the number of elements in the array segment $a[0 \ldots r][c]$ equal to x in some fixed column c. The function *cntLt* counts the number of elements in the array segment $a[0 \ldots a.length-1][c]$ smaller than x in the column c. As a consequence of these definitions, $pos(a[i][c],i,a,c)-1$ is the position of $a[i]$ in the sorted version of a.

The function *val* can easily be implemented in JML using the built-in constructs `sum` and `product`. The value of $cntEq(x,r,a,c)$ (and similarly $cntLt(x,a,c)$) can be represented in JML by:

```
\sum int i; 0<=i && i<=r; (x==a[i][c]) ? 1 : 0
```

19.2.2 Counting Sort Proof

With the above definitions in place, we are ready to prove that the implementation of Counting sort satisfies the contract in Listing 19.4. To this end, we devise the loop invariants of Counting sort. The first loop (Listing 19.1, lines 4–5) sets $c[i]$ to the number of occurrences of the value i in $a[0 \ldots j-1][m]$. Thus we use the invariant:

—— Java + JML ——

```
   0 <= j && j <= a.length
&& \forall int i; c[i] == cntEq(i, j-1, a, m);
```

—— Java + JML ——

The second loop replaces each $c[i]$ with its partial sum. We formalize this by the following invariant:

—— Java + JML ——

```
   1 <= j && j <= k
&& (\forall int i; 0 <= i && i <= j-1;
        c[i] == cntEq(i, a.length-1, a, m) + cntLt(i, a, m))
&& (\forall int i; j <= i && i < k;
        c[i] == cntEq(i, a.length-1, a, m));
```

—— Java + JML ——

The second conjunct ranges over the elements in c which have already been replaced by their partial sum. The third conjunct ranges over the elements which have not been processed yet (and hence, obey the postcondition of the first loop).

The invariant of the last loop is as follows:

—— Java + JML ——

```
   -1 <= j && j < a.length
&& (\forall int i; 0 <= i && i < a.length;
        c[a[i][m]] == pos(a[i][m], j, a, m))
&& (\forall int i; j+1 <= i && i < a.length;
        res[pos(a[i][m], i, a, m)-1] == a[i]);
```

—— Java + JML ——

Recall that $pos(a[i][m], i, a, m) - 1$ is the position of $a[i]$ in the sorted version of a. Thus the second conjunct intuitively means that $c[a[i][m]] - 1$ points to the position in which $a[i]$ should be stored in the sorted array. The assertion about *res* in the third conjunct expresses that *res* is the sorted version of a. This invariant gives rise to several proof obligations. We discuss the most interesting ones. For readability we abbreviate the invariant by I. Furthermore, whenever it is clear from the context we denote $pos(x, i, a, m)$ by $pos(x, i)$ and $pos(a[i][m], i)$ by $pos(i)$. Thus for example, $pos(i) - 1$ is the index of $a[i]$ in the sorted version of a.

Our first proof obligation states that *pos* obeys a weak form of injectivity.

$$\forall i \in [j : a.length - 1] : pos(i) = pos(j) \rightarrow a[j] = a[i]$$

This follows from the definitions of *pos*, *cntEq* and *cntLt*. The next verification condition characterizes the behavior of *pos*.

$$\begin{aligned}\forall i \in [0 : a.length - 1] :\ & a[i][m] = a[j][m] \rightarrow pos(a[j][m], j) - 1 = pos(a[i][m], j - 1) \\ & \wedge\ a[i][m] \neq a[j][m] \rightarrow pos(a[i][m], j) = pos(a[i][m], j - 1)\end{aligned}$$

The truth of the first conjunct follows from the fact that $cntEq(a[j][m], j, a, m) - 1 = cntEq(a[j][m], j-1, a, m)$. The second conjunct holds since $cntEq(x, j, a, m) = cntEq(x, j-1, a, m)$ whenever $x \neq a[j][m]$. The next verification condition states that after the execution of the loop (i.e., when $j = -1$), *res* must be sorted:

$$\forall i \in [0 : a.length - 2] : I \wedge j = -1 \rightarrow res[i][m] \leq res[i+1][m]$$

This is true since the invariant implies $res[pos(i) - 1] = a[i]$ for $i \in [0 : a.length - 1]$. But as remarked above, $pos(i) - 1$ is the position of $a[i]$ in the sorted version of a, hence *res* is sorted.

The final proof obligation concerns the proof of stability:

$$\begin{aligned}\forall r \in [0 : a.length - 2] : I \wedge j = -1 \wedge res[r][m] = res[r+1][m] \\ \rightarrow \exists i, j (0 \leq i < j < a.length) : res[r] = a[i] \wedge res[r+1] = a[j]\end{aligned}$$

Fix some arbitrary $r \in [0 : a.length - 2]$. We must show that $I \wedge j = -1 \wedge res[r][m] = res[r+1][m]$ implies $\exists i, j (0 \leq i < j < a.length) : res[r] = a[i] \wedge res[r+1] = a[j]$. Since the function $i \mapsto pos(i)$ is a bijection on $[1 : a.length]$ we must have $r = pos(i) - 1$ and $r + 1 = pos(j) - 1$ for some $i, j \in [0 : a.length - 1]$. Hence, only $i < j$ remains to show. This follows from the fact that $pos(i) < pos(j)$, together with the monotonicity property $pos(x, n) \leq pos(x, n+1)$ for all n (which follows from the earlier characterization of the behavior of *pos*). This proves the stability of our Counting sort implementation.

19.2.3 Radix Sort Proof

The correctness of Radix sort relies on the correctness of the stable sorting algorithm used in Radix sort. In the proof below, we assume only the contract of the generic stable sorting algorithm, instead of a particular implementation. This has the advantage that instead of being tied to Counting sort, *any* stable algorithm can be used within Radix sort, as long as it satisfies the contract of `stableSort`. Given the definitions of the auxiliary functions, the specification of Radix sort is as follows:

—— Java + JML ——

```
/*@ public normal_behavior
  @   requires
  @        k > 0
  @     && (\forall int j; 0 <= j && j < a.length;
  @             a[j].length == a[0].length)
```

```
  @     && (\forall int j,m;   0 <= j && j < a.length
  @                          && 0 <= m && m < a[j].length;
  @             0 <= a[j][m] && a[j][m] < k);
  @   ensures
  @       \dl_seqPerm(\dl_array2seq(\old(a)),
  @                   \dl_array2seq(\result))
  @     && (\forall int row; 0 <= row && row < a.length-1;
  @                 val(k,row,\old(a[0].length)-1,\result)
  @             <= val(k,row+1,\old(a[0].length)-1,\result));
  @*/
public static int[][] radixSort(int[][] a, int k);
```

Java + JML

The last conjunct in the precondition informally means that all digits that appear in a are nonnegative and bounded by k. The formula `\forall int row (...)` in the postcondition expresses that the large number in each row of the returned array is smaller or equal to the number in the next row, when interpreted in base k.

The correctness proof of Radix sort is based on the following loop invariant I:

JML

```
   0 <= i && i <= a[0].length && a != null
&& \dl_seqPerm(\dl_array2seq(a), \dl_array2seq(\old(a)))
&& (\forall int row; 0 <= row && row < a.length;
        \val(k,row,i-1,a) <= \val(k,row+1,i-1,a));
```

JML

Intuitively, the formula `\forall int row (...)` states that a is sorted with respect to the first i digits. When proving that the body of the loop preserves I, the main verification condition that arises states that the invariant follows from the postcondition of the procedure call, provided that the invariant was true initially. We refer to the contents of a before the call by introducing a logical variable A in the contract of `stableSort` as follows: we add $A = a$ to the precondition and substitute A for old(a) in the postcondition. Let $post'$ be the resulting postcondition. Formally the main verification condition is then as follows:

$$I[a := A] \wedge post'[\backslash result := a] \rightarrow \\ \forall row \in [0 : a.length - 1] : val(k, row, i, a) \leq val(k, row + 1, i, a)$$

To see why this formula is valid, consider an arbitrary row $r \in [0 : a.length - 2]$. Given the assumption $I[a := A] \wedge post'[\backslash result := a]$, we must prove $val(k,r,i,a) \leq val(k,r+1,i,a)$. From $post'[\backslash result := a]$ we infer $a[r][i] \leq a[r+1][i]$. We distinguish two cases.

- $a[r][i] < a[r+1][i]$. Then also $a[r][i] * k^i < a[r+1][i] * k^i$. Clearly $val(k,r,i-1,a) < k^i$, since $val(k,r,i-1,a)$ is a number with i digits in base k, while k^i has $i+1$ digits in base k. But $val(k,r,i,a) = val(k,r,i-1,a) + a[r][i] * k^i$, hence $val(k,r,i,a) \leq val(k,r+1,i,a)$.

- $a[r][i] = a[r+1][i]$. Then it suffices to prove $val(k,r,i-1,a) \leq val(k,r+1,i-1,a)$. But $post'[\backslash result := a]$ implies that $a[r] = A[m]$ and $a[r+1] = A[n]$ for some m,n (with $0 \leq m < n < a.length$), so it suffices to prove $val(k,m,i-1,A) \leq val(k,n,i-1,A)$. But the invariant implies $val(k,r_1,i-1,A) \leq val(k,r_2,i-1,A)$ if $r_1 < r_2$. Instantiating r_1 with m and r_2 with n gives the desired result.

This concludes the proof of Radix sort.

19.3 Experience Report

In this section we discuss our practical experience with KeY. The following table summarizes some statistics of the proofs in KeY:

	Counting Sort	Radix Sort
Rule applications	96.260	114.309
User interactions	743 (0.8%)	762 (0.7%)

"Rule applications" serves as a measure for the length of the proofs: this row contains the total number of proof rule applications used in the proofs, whereas "User interactions" indicates the number of proof rule applications that were applied manually by the authors (i.e., required creativity). The statistics show that the degree of automation of KeY for both algorithms was over 99%.

The mechanized proofs are significantly larger than the high-level proofs, for several reasons. First, we used the automatic proof strategies of KeY as much as possible, but the strategies do not always find the shortest proofs. Second, in the actual mechanized proofs we also showed termination. Fortunately this did not require much creativity: the ranking functions (loop variants) are trivial to find and prove since all loops that occur are `for`-loops. After appropriate ranking functions were given, the proof of termination was automatic. A third reason for the large proofs is that Java has several features that were ignored in the high-level proofs but complicate the mechanized KeY proofs.

One such Java feature is the fact that arrays are bounded. For example, to ensure that the assignment `res[c[a[j]]] = a[j];` does not lead to index out-of-bounds exceptions, KeY generates *four* proof obligations: j must be within the bounds of the array a (this condition must be proven twice, since $a[j]$ occurs twice), and $a[j]$ and $c[a[j]]$ must be within the array bounds of respectively C and res. This duplication of proofs, caused by multiple references to the same array element, could be avoided by changing the Java source to `int tmp = a[j]; res[c[tmp]] = tmp;`. KeY was able to automatically prove that the array references to c in the first two loops did not violate the array bounds, and similarly for a in the third loop. The references to res and c in the third loop required some user interactions. In particular, it required proving that $1 \leq pos(i) \leq a.length$. Still, overall, less than 5% of the rule applications concerned array bounds.

The part of the proof by far responsible for the most rule applications (over 60%!) surprisingly is unrelated to deriving validity of the verification conditions discussed

in the previous section. Instead it concerns proving that the value of our auxiliary functions *cntEq*, *cntLt*, *pos* and *val* is the same in different heaps that arise during execution (despite using the tightest possible assignable clauses in all loops and contracts). Note first that these functions indeed depend on the contents of the heap, since their value depends on the contents of an array (the array object *a* passed as a parameter), and arrays are allocated on the heap. Since the heap is represented explicitly by a term in KeY, the actual KeY formalization of the definitions of these functions contain an additional parameter *Heap*. In fact, since *a* is the only parameter of the auxiliary functions which has a class type, the value of the auxiliary functions depends *only* on the part of the heap containing *a*; other parts of the heap are simply not visible. However, the program never changes the contents of *a*: only parts of the heap irrelevant to the value of the auxiliary functions are changed. Unfortunately, currently KeY cannot detect this, nor can the user specify it, without unrolling the definition of the auxiliary functions. After unrolling, KeY could prove in most (but not all) cases automatically that the heap was only changed in ways irrelevant to the value of the auxiliary functions, though at the expense of a huge number of rule applications due to the size of the involved heap terms. One partial workaround for this is to surround any reference to the auxiliary functions by old in the loop invariants. This seemingly small change, which causes all occurrences of auxiliary functions to be evaluated in the same (old) heap, resulted in a reduction of the Radix sort proof from 169.030 rule applications to a little over the current 114.309! This change also reduced the number of manual user interactions by about 30%. A further potential improvement would be to use model methods that return the value of the auxiliary functions (instead of using the auxiliary functions directly), as the user can specify an `accessible` clause for model methods (see Section 7.9.1).

One final discussion point concerns the permutation predicate *seqPerm*. The detailed JavaDL formalization of this predicate and the sequence data type can be found in Chapter 5. The sequence data type and corresponding permutation predicates have been newly added to KeY 2.x but so far, little was known about the implications regarding automation. The present case study provides some empirical results: about 20% of the total manual interactions concerned reasoning about sequences.