View other issues

Contents: ERCIM News 102

Fixing the Sorting Algorithm for Android, Java and Python

by Stijn de Gouw and Frank de Boer

In 2013, whilst trying to prove the correctness of TimSort - a broadly applied sorting algorithm - the CWI Formal Methods Group, in collaboration with SDL, Leiden University and TU Darmstadt, instead identified an error in it, which could crash programs and threaten security. Our bug report with an improved version, developed in February 2015, has led to corrected versions in major programming languages and frameworks.

Tim Peters developed the Timsort hybrid sorting algorithm in 2002. TimSort was first developed for Python, a popular programming language, but later ported to Java (where it appears as java.util.Collections.sort and java.util.Arrays.sort). TimSort is today used as the default sorting algorithm in Java, in Android (a widely used platform by Google for mobile devices), in Python and many other programming languages and frameworks. Given the popularity of these platforms this means that the number of computers, cloud services and mobile phones that use TimSort for sorting is well into the billions.

After we had successfully verified Counting and Radix sort implementations in Java [1] with a formal verification tool called KeY, we were looking for a new challenge. TimSort seemed to fit the bill, as it is rather complex and widely used. Unfortunately, we weren't able to prove its correctness. A closer analysis showed that this was, quite simply, because TimSort was broken and our theoretical considerations finally led us to a path towards finding the bug (interestingly, that bug appears already in the Python implementation). Here we sketch how we did it.

TimSort reorders the input array from left to right by finding consecutive (disjoint) sorted segments (called "runs" from here on). The lengths of the generated runs are added to an array named *runLen*. Whenever a new run is added to *runLen*, a method named mergeCollapse merges runs until the last 3 elements in *runLen* satisfies certain conditions , the most important one being
runLen[n-2] > runLen[n-1] + runLen[n].

This condition says that the sum of the last two runs is strictly smaller than the third last run and follows the pattern of the well-known Fibonacci sequence. The intention is that checking this condition on the top 3 runs in *runLen* in fact guarantees that all runs satisfy it (the "invariant"). At the very end, all runs are merged, yielding a sorted version of the input array.

For performance reasons, it is crucial to allocate as little memory as possible for *runLen*, but still enough to store all the runs. If the invariant is satisfied, the run lengths in reverse order grow exponentially (even faster than the Fibonacci sequence: the length of the current run must be strictly bigger than the sum of the next two runs lengths). Since runs do not overlap, only a small number of runs would then be needed to cover even very big input arrays completely.

However, when we tried to prove the invariant formally, we found out that it is not sufficient to check only the top 3 runs in *runLen*. We developed a test generator that builds an input array with many short runs – too short, in the sense that they break the invariant – which eventually causes TimSort to crash with an ArrayOutOfBoundsException.

We also succeeded to fix TimSort by checking the last 4 runs and formally verify this new version using a deductive verification platform for sequential Java and JavaCard applications, called KeY. It allows to statically prove the correctness of programs for any given input with respect to a given specification. Roughly speaking, a specification consists of a precondition (a condition on the input), also called requires clause and a postcondition (a condition on the output), also called ensures clause. Specifications are attached to method implementations, such as mergeCollapse() above.

The (simplified) mergeCollapse contract (Figure 1) illustrates these concepts.

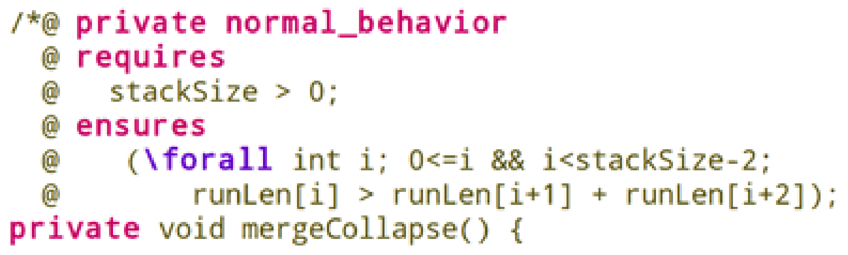
```
/*@ private normal_behavior
  @ requires
  @   stackSize > 0;
  @ ensures
  @    (\forall int i; 0<=i && i<stackSize-2;
  @         runLen[i] > runLen[i+1] + runLen[i+2]);
private void mergeCollapse() {
```

Figure 1: The (simplified) mergeCollapse contract.

The precondition *stackSize > 0* means intuitively that *mergeCollapse()* should only be called when at least one run has been added. The two formulas in the postcondition (ensures) imply that after mergeCollapse completes, all runs satisfy the invariant. Without tool support and automated theorem proving technology it is hardly possible to come up with correct invariants for non-trivial programs. And in fact, it is exactly here that the designers of TimSort went wrong.
So far, this was one of the hardest correctness proofs ever of an existing Java library. It required more than two million rules of inference and thousands of manual steps. With such an widely used language like Java, it is important that software does not crash. This result illustrates the relevance of formal methods, e.g., in Python our fix was quickly applied.

Other recent successful applications of formal methods are INFER, an automatic, separation-logic-based memory safety checker used in Facebook and the Temporal Logic of Actions (TLA). TLA is developed by Leslie Lamport, Recipient of the Turing Award 2013. It is in use by engineers at Amazon Web Services.The work was co-funded by the EU project Envisage.

Link:
http://envisage-project.eu/wp-content/uploads/2015/02/sorting.pdf

Reference:
S. de Gouw, F. de Boer, J. Rot: “Proof Pearl: The KeY to Correct and Stable Sorting”, Journal of Autom. Reasoning 53(2), 129-139, 2014.

Please contact:
Stijn de Gouw, Frank de Boer, CWI, The Netherlands
E-mail: cdegouw@cwi.nl, f.s.de.boer@cwi.nl