Showing that Android's, Java's and Python's sorting algorithm is broken and fixing it formally

Stijn de Gouw
Jurriaan Rot, Frank de Boer, Richard Bubel, Reiner Hähnle

CWI Amsterdam / SDL Fredhopper

Formal Methods 2015
Oslo, June 22, 2015

http://www.envisage-project.eu

Programming languages: Libraries

Library

Collection of commonly used algorithms that are invoked through a well-defined interface

Programming languages: Libraries

Library

Collection of commonly used algorithms that are invoked through a well-defined interface

Example: Java standard library functions

Programming to interfaces:

- Sorting a given array `a`

  ```
  static void sort(Object[] a)
  ```

- Searching a value `key` in the array `a`

  ```
  static int binarySearch(Object[] a, Object key)
  ```

Usability of programming language partially depends on good libraries

Programming languages: Libraries

Library

Collection of commonly used algorithms that are invoked through a well-defined interface

Example: Java standard library functions

Programming to interfaces:

- Sorting a given array `a`

  ```
  static void sort(Object[] a)
  ```

- Searching a value `key` in the array `a`

  ```
  static int binarySearch(Object[] a, Object key)
  ```

Usability of programming language partially depends on good libraries

Correctness of library functions is crucial:
used as building blocks in millions of programs

Description

Timsort: a hybrid sorting algorithm (insertion sort + merge sort) optimized for partially sorted arrays (often encountered in real-world data).

Timsort (I)

Description

Timsort: a hybrid sorting algorithm (insertion sort + merge sort) optimized for partially sorted arrays (often encountered in real-world data).

Timsort is used in

- Java (standard library), used by Oracle
- Python (standard library), used by Google
- Android (standard library), used by Google
- Hadoop (Big data), used by Apache, Facebook and Yahoo
- ... and many more languages / frameworks!

`TimSort.rangeCheck` appeared in court case between Oracle and Google

Timsort (I)

Description

Timsort: a hybrid sorting algorithm (insertion sort + merge sort) optimized for partially sorted arrays (often encountered in real-world data).

Timsort is used in

- Java (standard library), used by Oracle
- Python (standard library), used by Google
- Android (standard library), used by Google
- Hadoop (Big data), used by Apache, Facebook and Yahoo
- ... and many more languages / frameworks!

`TimSort.rangeCheck` appeared in court case between Oracle and Google

Why analyze Timsort?

- Complex algorithm, widely used
- Extensively tested + manual code reviews: bugs unlikely!?

Timsort (II)

The algorithm

- Find next already sorted segment ("runs") extending to length ≥ 16 with insertion sort.
- Add length of new run to `runLen` array
- Merge until last 3 runs satisfy two conditions ("the invariant")
  1. `runLen[n-2] > runLen[n-1] + runLen[n]`
  2. `runLen[n-1] > runLen[n]`

  Merging: if (1) is false and `runLen[n-2] < runLen[n]`, merge runs at `n-2` and `n-1`, otherwise at `n-1` and `n`
- At the end: merge all runs, resulting in a sorted array

Example, ignoring length ≥ 16 requirement

Input										
	1	2	3	4	5	0	1	1	0	4

runLen

Timsort (II)

The algorithm

- **Find next already sorted segment ("runs") extending to length ≥ 16 with insertion sort.**
- **Add length of new run to `runLen` array**
- Merge until last 3 runs satisfy two conditions ("the invariant")
  1. `runLen[n-2] > runLen[n-1] + runLen[n]`
  2. `runLen[n-1] > runLen[n]`

  Merging: if (1) is false and `runLen[n-2] < runLen[n]`, merge runs at `n-2` and `n-1`, otherwise at `n-1` and `n`
- At the end: merge all runs, resulting in a sorted array

Example, ignoring length ≥ 16 requirement

Input	1	2	3	4	5	0	1	1	0	4
runLen	5									

Timsort (II)

The algorithm

- **Find next already sorted segment ("runs") extending to length ≥ 16 with insertion sort.**
- **Add length of new run to `runLen` array**
- Merge until last 3 runs satisfy two conditions ("the invariant")
  1. `runLen[n-2] > runLen[n-1] + runLen[n]`
  2. `runLen[n-1] > runLen[n]`

  Merging: if (1) is false and `runLen[n-2] < runLen[n]`, merge runs at `n-2` and `n-1`, otherwise at `n-1` and `n`
- At the end: merge all runs, resulting in a sorted array

Example, ignoring length ≥ 16 requirement

Input	1	2	3	4	5	0	1	1	0	4

runLen	5	3

Timsort (II)

The algorithm

- **Find next already sorted segment ("runs") extending to length ≥ 16 with insertion sort.**
- **Add length of new run to `runLen` array**
- Merge until last 3 runs satisfy two conditions ("the invariant")
  1. `runLen[n-2] > runLen[n-1] + runLen[n]`
  2. `runLen[n-1] > runLen[n]`

  Merging: if (1) is false and `runLen[n-2] < runLen[n]`, merge runs at `n-2` and `n-1`, otherwise at `n-1` and `n`
- At the end: merge all runs, resulting in a sorted array

Example, ignoring length ≥ 16 requirement

Input	1	2	3	4	5	0	1	1	0	4

runLen	5	3	2

Timsort (II)

The algorithm

- Find next already sorted segment ("runs") extending to length ≥ 16 with insertion sort.
- Add length of new run to `runLen` array
- Merge until last 3 runs satisfy two conditions ("the invariant")
  1. **`runLen[n-2] > runLen[n-1] + runLen[n]`**
  2. `runLen[n-1] > runLen[n]`

  Merging: if **(1) is false** and `runLen [n-2] < runLen [n]`, merge runs at `n-2` and `n-1`, **otherwise at `n-1` and `n`**
- At the end: merge all runs, resulting in a sorted array

Example, ignoring length ≥ 16 requirement

Input	1	2	3	4	5	0	1	1	0	4

runLen	5	3	2

Timsort (II)

The algorithm

- Find next already sorted segment ("runs") extending to length ≥ 16 with insertion sort.
- Add length of new run to `runLen` array
- Merge until last 3 runs satisfy two conditions ("the invariant")
  1. `runLen[n-2] > runLen[n-1] + runLen[n]`
  2. `runLen[n-1] > runLen[n]`

  Merging: if (1) is false and `runLen[n-2] < runLen[n]`, merge runs at `n-2` and `n-1`, otherwise at `n-1` and `n`
- At the end: merge all runs, resulting in a sorted array

Example, ignoring length ≥ 16 requirement

Input

1	2	3	4	5	0	0	1	1	4

runLen

5	5

Timsort (II)

The algorithm

- Find next already sorted segment ("runs") extending to length ≥ 16 with insertion sort.
- Add length of new run to `runLen` array
- Merge until last 3 runs satisfy two conditions ("the invariant")
  1. `runLen[n-2] > runLen[n-1] + runLen[n]`
  2. **`runLen[n-1] > runLen[n]`**

  Merging: if (1) is false and `runLen[n-2] < runLen[n]`, merge runs at `n-2` and `n-1`, **otherwise at `n-1` and `n`**
- At the end: merge all runs, resulting in a sorted array

Example, ignoring length ≥ 16 requirement

Input	1	2	3	4	5	0	0	1	1	4
runLen	5	5								

Timsort (II)

The algorithm

- Find next already sorted segment ("runs") extending to length ≥ 16 with insertion sort.
- Add length of new run to `runLen` array
- Merge until last 3 runs satisfy two conditions ("the invariant")
  1. `runLen[n-2] > runLen[n-1] + runLen[n]`
  2. `runLen[n-1] > runLen[n]`

  Merging: if (1) is false and `runLen[n-2] < runLen[n]`, merge runs at `n-2` and `n-1`, otherwise at `n-1` and `n`
- At the end: merge all runs, resulting in a sorted array

Example, ignoring length ≥ 16 requirement

Input	0	0	1	1	1	2	3	4	4	5
runLen	10									

Timsort (II)

The algorithm

- Find next already sorted segment (“runs”) extending to length ≥ 16 with insertion sort.
- Add length of new run to `runLen` array
- Merge until last 3 runs satisfy two conditions (… riant”)
  1. `runLen[n-2] > runLen[n-1] + runLen[n]`
  2. `runLen[n-1] > runLen[n]`

  Merging: if (1) is false and `runLen[n-2] < runLen[n]`, merge runs at `n-2` and `n-1`, otherwise at `n-1` and `n`
- At the end: merge all runs, resulting in a sorted array

Fibonacci ?!

Example, ignoring length ≥ 16 requirement

Input	0	0	1	1	1	2	3	4	4	5

runLen	10

Breaking the invariant

Size of `runLen`

1. `runLen[n-2] > runLen[n-1] + runLen[n]`
2. `runLen[n-1] > runLen[n]`

If the above invariant is true for all *n* and `runLen[n] >= 16`, then

- (reversed) runlengths grow exponentially fast (... 87 52 34 17 16)
- Runs do not overlap: few runs required to cover input array

Breaking the invariant

Size of `runLen`

1. `runLen[n-2] > runLen[n-1] + runLen[n]`
2. `runLen[n-1] > runLen[n]`

If the above invariant is true for all *n* and `runLen[n] >= 16`, then

- (reversed) runlengths grow exponentially fast (... 87 52+34+17+16)
- Runs do not overlap: few runs required to cover input array

```
int stackLen = (len <    120 ?  4 :
                len <   1542  ?  9 :
                len < 119151  ? 18 : 39);
runBase = new int[stackLen];
runLen = new int[stackLen];
```

Breaking the invariant

Size of `runLen`

1. `runLen[n-2] > runLen[n-1] + runLen[n]`
2. `runLen[n-1] > runLen[n]`

If the above invariant is true for all *n* and `runLen[n] >= 16`, then

- (reversed) runlengths grow exponentially fast (... 87 52 34 17 16)
- Runs do not overlap: few runs required to cover input array

Breaking the invariant

Size of `runLen`

1. `runLen[n-2] > runLen[n-1] + runLen[n]`
2. `runLen[n-1] > runLen[n]`

If the above invariant is true for all *n* and `runLen[n] >= 16`, then

- (reversed) runlengths grow exponentially fast (... 87 52 34 17 16)
- Runs do not overlap: few runs required to cover input array

Breaking the invariant - checking last 3 runs is insufficient

If (1) is false and `runLen [n-2] < runLen [n]`: merge at idx `n-2` and `n-1`, otherwise merge runs at indices `n-1` and `n`

runLen				
	120	80	25	20

Breaking the invariant

Size of `runLen`

1. `runLen[n-2] > runLen[n-1] + runLen[n]`
2. `runLen[n-1] > runLen[n]`

If the above invariant is true for all *n* and `runLen[n] >= 16`, then

- (reversed) runlengths grow exponentially fast (... 87 52 34 17 16)
- Runs do not overlap: few runs required to cover input array

Breaking the invariant - checking last 3 runs is insufficient

If (1) is false and `runLen [n-2] < runLen [n]`: merge at idx `n-2` and `n-1`, otherwise merge runs at indices `n-1` and `n`

runLen					
runLen	120	80	25	20	30

Breaking the invariant

Size of `runLen`

1. `runLen[n-2] > runLen[n-1] + runLen[n]`
2. `runLen[n-1] > runLen[n]`

If the above invariant is true for all *n* and `runLen[n] >= 16`, then

- (reversed) runlengths grow exponentially fast (... 87 52 34 17 16)
- Runs do not overlap: few runs required to cover input array

Breaking the invariant - checking last 3 runs is insufficient

If (1) is false and `runLen [n-2] < runLen [n]`: merge at idx `n-2` and `n-1`, otherwise merge runs at indices `n-1` and `n`

runLen					
	120	80	25	20	30

Breaking the invariant

Size of `runLen`

1. `runLen[n-2] > runLen[n-1] + runLen[n]`
2. `runLen[n-1] > runLen[n]`

If the above invariant is true for all *n* and `runLen[n] >= 16`, then

- (reversed) runlengths grow exponentially fast (... 87 52 34 17 16)
- Runs do not overlap: few runs required to cover input array

Breaking the invariant - checking last 3 runs is insufficient

If (1) is false and `runLen [n-2] < runLen [n]`: merge at idx `n-2` and `n-1`, otherwise merge runs at indices `n-1` and `n`

runLen				
	120	80	45	30

Breaking the invariant

Size of `runLen`

1. `runLen[n-2] > runLen[n-1] + runLen[n]`
2. `runLen[n-1] > runLen[n]`

If the above invariant is true for all *n* and `runLen[n] >= 16`, then

- (reversed) runlengths grow exponentially fast (... 87 52 34 17 16)
- Runs do not overlap: few runs required to cover input array

Breaking the invariant - checking last 3 runs is insufficient

If (1) is false and `runLen [n-2] < runLen [n]`: merge at idx `n-2` and `n-1`, otherwise merge runs at indices `n-1` and `n`

runLen				
	120	80	45	30

Our work (I)

Wrote program that generates testcase

- that exploits breaking the invariant, by generating too many "short" runs
- Triggers exception: insufficient size for `runLen` to store run lengths

Language	Smallest array that triggers error
Android	65.536 (2^{16})
Java	67.108.864 (2^{26})
Python	562.949.953.421.312 (2^{49})

Most powerful supercomputer (Tianhe-2) has $\approx 2^{50}$ bytes of mem.

Our work (I)

Wrote program that generates testcase

- that exploits breaking the invariant, by generating too many "short" runs
- Triggers exception: insufficient size for `runLen` to store run lengths

Language	Smallest array that triggers error
Android	65.536 (2^{16})
Java	67.108.864 (2^{26})
Python	562.949.953.421.312 (2^{49})

Most powerful supercomputer (Tianhe-2) has $\approx 2^{50}$ bytes of mem.

Provided worst-case analysis of broken version

- Shows the actual minimally required `runLen.length`

Our work (II)

Fixed the algorithm

- Check that last 4 runs satisfy invariant
- Executed existing benchmarks (result: same performance) and unit tests (all passed)

```
/** ...
  * merges adjacent runs until the stack invariants are reestablished:
  *     1. runLen[i - 3] > runLen[i - 2] + runLen[i - 1]
  *     2. runLen[i - 2] > runLen[i - 1]
  */
private void mergeCollapse() {
    while (stackSize > 1) {
        int n = stackSize - 2;
        if (    (n >= 1 && runLen[n-1] <= runLen[n] + runLen[n+1])
             || (n >= 2 && runLen[n-2] <= runLen[n-1] + runLen[n]) ){
            if (runLen[n - 1] < runLen[n + 1])
                n--;
        } else if (runLen[n] > runLen[n + 1]) {
            break; // Invariant is established
        }
        mergeAt(n);
    }
}
```

Our Work (III)

Analyzing "Real" Software

> *"because truly understanding it essentially requires doing a formal correctness proof, it's difficult to maintain"*

> *"Yet another large mass of difficult code can make for a real maintenance burden after I'm dead"*

- Tim Peters on Timsort, python-dev mailing list, 2002

Implementation uses features for performance that complicate analysis: break statements, low-level bitwise ops., arithmetic overflows

Our Work (III)

Analyzing "Real" Software

> *"because truly understanding it essentially requires doing a formal correctness proof, it's difficult to maintain"*

> *"Yet another large mass of difficult code can make for a real maintenance burden after I'm dead"*

- Tim Peters on Timsort, python-dev mailing list, 2002

Implementation uses features for performance that complicate analysis: break statements, low-level bitwise ops., arithmetic overflows

Mechanically proved fixed version with KeY (Java theorem prover)

- absence of the bug, and all other run-time exceptions
- termination
- this requires: formal specifications for all functions

Specifying Java Code with JML

Method contracts

- precondition (`requires`): condition on the input
- postcondition (`ensures`): condition on the output / result

```
/*@ requires
  @   stackSize > 0;
  @ ensures
  @   (\forall int i; 0<=i && i<stackSize-2;
  @          elemInv(runLen, i, 16))
  @   && elemBiggerThanNext(runLen, stackSize-2)
  @*/
private void mergeCollapse()
```

Specifying Java Code with JML

Method contracts

- precondition (`requires`): condition on the input
- postcondition (`ensures`): condition on the output / result

```
/*@ requires
  @   stackSize > 0;
  @ ensures
  @   (\forall int i; 0<=i && i<stackSize-2;
  @          elemInv(runLen, i, 16))
  @   && elemBiggerThanNext(runLen, stackSize-2)
  @*/
private void mergeCollapse()
```

Class Invariant

Property that all instances of a class must satisfy before and after every method (call)

- Can be assumed in method precondition
- Must be established at all call sites and method postcondition

Class Invariant (simplified)

```
/*@ invariant
@       runBase.length == runLen.length
@ && (a.length  <    120                    ==> runLen.length==4)
@ && (a.length >=    120 && a.length < 1542 ==> runLen.length==9)
@ && (a.length >=   1542 && a.length<119151 ==> runLen.length==18)
@ && (a.length >= 119151                    ==> runLen.length==39)
@ && (0 <= stackSize && stackSize <= runLen.length)
@ && (\forall int i; 0<=i && i<stackSize-4;
@       elemInv(runLen, i, 16))
@ && (elemLargerThanBound(runBase, 0, 0))
@ && (\forall int i; 0<=i && i<stackSize-1;
@        runBase[i] + runLen[i] == runBase[i+1]);
@*/
```

Name	Definition
elemBiggerThanNext2(*arr*, *idx*)	$(0 \leq idx \wedge idx + 2 < arr.length) \rightarrow$ $arr[idx] > arr[idx + 1] + arr[idx + 2]$
elemBiggerThanNext(*arr*, *idx*)	$0 \leq idx \wedge idx + 1 < arr.length \rightarrow$ $arr[idx] > arr[idx + 1]$
elemLargerThanBound(*arr*, *idx*, *v*)	$0 \leq idx < arr.length \rightarrow$ $arr[idx] \geq v$
elemInv(*arr*, *idx*, *v*)	elemBiggerThanNext2(*arr*, *idx*)$\wedge$ elemBiggerThanNext(*arr*, *idx*)$\wedge$ elemLargerThanBound(*arr*, *idx*, *v*)

Class Invariant (simplified)

```
/*@ invariant
@        runBase.length == runLen.length
@ && (a.length < 120 ==> runLen.length==4)
@ && (a.length >= 120 && a.length < 1542 ==> runLen.length==9)
@ && (a.length >= 1542 && a.length<119151 ==> runLen.length==18)
@ && (a.length >= 119151 ==> runLen.length==39)
@ && (0 <= stackSize && stackSize <= runLen.length)
@ && (\forall int i; 0<=i && i<stackSize-4;
@        elemInv(runLen, i, 16))
@ && (elemLargerThanBound(runBase, 0, 0))
@ && (\forall int i; 0<=i && i<stackSize-1;
@         runBase[i] + runLen[i] == runBase[i+1]);
@*/
```

Length of `runlen` in terms of input length

Class Invariant (simplified)

```
/*@ invariant
@        runBase.length == runLen.length
@ && (a.length  <    120                    ==> runLen.length==4)
@ && (a.length >=    120 && a.length < 1542 ==> runLen.length==9)
@ && (a.length >=   1542 && a.length<119151 ==> runLen.length==18)
@ && (a.length >= 119151                    ==> runLen.length==39)
@ && (0 <= stackSize && stackSize <= runLen.length)
@ && (\forall int i; 0<=i && i<stackSize-4;
@        elemInv(runLen, i, 16))
@ && (elemLargerThanBound(runBase, 0, 0))
@ && (\forall int i; 0<=i && i<stackSize-1;
@         runBase[i] + runLen[i] == runBase[i+1]);
@*/
```

Bounds on stackSize (in-use part of `runLen`)

Class Invariant (simplified)

```
/*@ invariant
@       runBase.length == runLen.length
@ && (a.length  <    120                           ==> runLen.length==4)
@ && (a.length >=    120 && a.length < 1542 ==> runLen.length==9)
@ && (a.length >=   1542 && a.length<119151 ==> runLen.length==18)
@ && (a.length >= 119151                          ==> runLen.length==39)
@ && (0 <= stackSize && stackSize <= runLen.length)
@ && (\forall int i; 0<=i && i<stackSize-4;
@        elemInv(runLen, i, 16))
@ && (elemLargerThanBound(runBase, 0, 0))
@ && (\forall int i; 0<=i && i<stackSize-1;
@        runBase[i] + runLen[i] == runBase[i+1]);
@*/
```

All but the last 4 runs satisfy the invariant while merging

Class Invariant (simplified)

```
/*@ invariant
@       runBase.length == runLen.length
@ && (a.length  <    120                          ==> runLen.length==4)
@ && (a.length >=    120 && a.length < 1542 ==> runLen.length==9)
@ && (a.length >=   1542 && a.length<119151 ==> runLen.length==18)
@ && (a.length >= 119151                          ==> runLen.length==39)
@ && (0 <= stackSize && stackSize <= runLen.length)
@ && (\forall int i; 0<=i && i<stackSize-4;
@       elemInv(runLen, i, 16))
@ && (elemLargerThanBound(runBase, 0, 0))
@ && (\forall int i; 0<=i && i<stackSize-1;
@        runBase[i] + runLen[i] == runBase[i+1]);
@*/
```

First run starts at non-negative array index

Class Invariant (simplified)

```
/*@ invariant
@        runBase.length == runLen.length
@ && (a.length  <    120                       ==> runLen.length==4)
@ && (a.length >=    120 && a.length < 1542 ==> runLen.length==9)
@ && (a.length >=   1542 && a.length<119151 ==> runLen.length==18)
@ && (a.length >= 119151                       ==> runLen.length==39)
@ && (0 <= stackSize && stackSize <= runLen.length)
@ && (\forall int i; 0<=i && i<stackSize-4;
@        elemInv(runLen, i, 16))
@ && (elemLargerThanBound(runBase, 0, 0))
@ && (\forall int i; 0<=i && i<stackSize-1;
@        runBase[i] + runLen[i] == runBase[i+1]);
@*/
```

There are no gaps between consecutive runs

mergeCollapse proof

Loop Invariant (simplified)

```
/*@ loop_invariant
  @  \forall int i; 0<=i && i<stackSize-4;
  @       elemInv(runLen, i, 16);
  @*/
```

The main verif. condition (simplified)

```
(  loop-inv && n==stackSize-2 && n >= 0
 && n>=1 ==>  runLen[n-1] > runLen[n] + runLen[n+1]
 && n>=2 ==>  runLen[n-2] > runLen[n-1] + runLen[n]
 && runLen[n] > runLen[n+1]
)      ==> ensures(mergeCollapse)
```

Recall that `ensures(mergeCollapse)` is (substituting `stackSize-2==n`):

```
(\forall int i; 0<=i && i<n; elemInv(runLen, i, 16))
&& elemBiggerThanNext(runLen, n)
```

pushRun contract (simplified)

```
/*@ normal_behavior
@ requires
@     (runLen > 0 && runBase >= 0)
@ && (stackSize > 0  ==> runBase ==
@     this.runBase[stackSize-1]+this.runLen[stackSize-1])
@ && (runLen + runBase <= a.length)
@ && (\forall int i; 0<=i && i<stackSize-2;
@          elemInv(this.runLen,i,16))
@ && elemBiggerThanNext(this.runLen, stackSize-2)
@ && elemLargerThanBound(this.runLen, stackSize-1, 16)
@ ensures
@     this.runBase[\old(stackSize)] == runBase
@ && this.runLen[\old(stackSize)] == runLen
@ && stackSize == \old(stackSize)+1;
@*/
private void pushRun(int runBase, int runLen) {
  this.runBase[stackSize] = runBase;
  this.runLen[stackSize] = runLen;
  stackSize++;
}
```

pushRun contract (simplified)

```
/*@ normal_behavior
@ requires
@ (runLen > 0 && runBase >= 0)
@ && (stackSize > 0 ==> runBase ==
@     this.runBase[stackSize-1]+this.runLen[stackSize-1])
@ && (runLen + runBase <= a.length)
@ && (\forall int i; 0<=i && i<stackSize-2;
@          elemInv(this.runLen,i,16))
@ && elemBiggerThanNext(this.runLen, stackSize-2)
@ && elemLargerThanBound(this.runLen, stackSize-1, 16)
@ ensures
@    this.runBase[\old(stackSize)] == runBase
@ && this.runLen[\old(stackSize)] == runLen
@ && stackSize == \old(stackSize)+1;
@*/
private void pushRun(int runBase, int runLen) {
  this.runBase[stackSize] = runBase;
  this.runLen[stackSize] = runLen;
  stackSize++;
}
```

The new run has positive length and starts directly after the last run

pushRun contract (simplified)

```
/*@ normal_behavior
@ requires
@     (runLen > 0 && runBase >= 0)
@ && (stackSize > 0  ==> runBase ==
@    this.runBase[stackSize-1]+this.runLen[stackSize-1])
@ && (runLen + runBase <= a.length)
@ && (\forall int i; 0<=i && i<stackSize-2;
@           elemInv(this.runLen,i,16))
@ && elemBiggerThanNext(this.runLen, stackSize-2)
@ && elemLargerThanBound(this.runLen, stackSize-1, 16)
@ ensures
@    this.runBase[\old(stackSize)] == runBase
@ && this.runLen[\old(stackSize)] == runLen
@ && stackSize == \old(stackSize)+1;
@*/
private void pushRun(int runBase, int runLen) {
  this.runBase[stackSize] = runBase;
  this.runLen[stackSize] = runLen;
  stackSize++;
}
```

The new run cannot extend beyond length of the input array

pushRun contract (simplified)

```
/*@ normal_behavior
@ requires
@     (runLen > 0 && runBase >= 0)
@ && (stackSize > 0  ==> runBase ==
@     this.runBase[stackSize-1]+this.runLen[stackSize-1])
@ && (runLen + runBase <= a.length)
@ && (\forall int i; 0<=i && i<stackSize-2;
@           elemInv(this.runLen,i,16))
@ && elemBiggerThanNext(this.runLen, stackSize-2)
@ && elemLargerThanBound(this.runLen, stackSize-1, 16)
@ ensures
@     this.runBase[\old(stackSize)] == runBase
@ && this.runLen[\old(stackSize)] == runLen
@ && stackSize == \old(stackSize)+1;
@*/
private void pushRun(int runBase, int runLen) {
  this.runBase[stackSize] = runBase;
  this.runLen[stackSize] = runLen;
  stackSize++;
}
```

The invariant is satisfied by all runs

pushRun contract (simplified)

```
/*@ normal_behavior
@ requires
@     (runLen > 0 && runBase >= 0)
@ && (stackSize > 0  ==> runBase ==
@     this.runBase[stackSize-1]+this.runLen[stackSize-1])
@ && (runLen + runBase <= a.length)
@ && (\forall int i; 0<=i && i<stackSize-2;
@          elemInv(this.runLen,i,16))
@ && elemBiggerThanNext(this.runLen, stackSize-2)
@ && elemLargerThanBound(this.runLen, stackSize-1, 16)
@ ensures
@     this.runBase[\old(stackSize)] == runBase
@ && this.runLen[\old(stackSize)] == runLen
@ && stackSize == \old(stackSize)+1;
@*/
private void pushRun(int runBase, int runLen) {
  this.runBase[stackSize] = runBase;
  this.runLen[stackSize] = runLen;
  stackSize++;
}
```

The new run is stored at index stackSize-1

pushRun main verification condition

No ArrayIndexOutOfBoundsException if

`requires(pushRun) &&` cl. invariant `==> stackSize < len.length`

pushRun main verification condition

No ArrayIndexOutOfBoundsException if

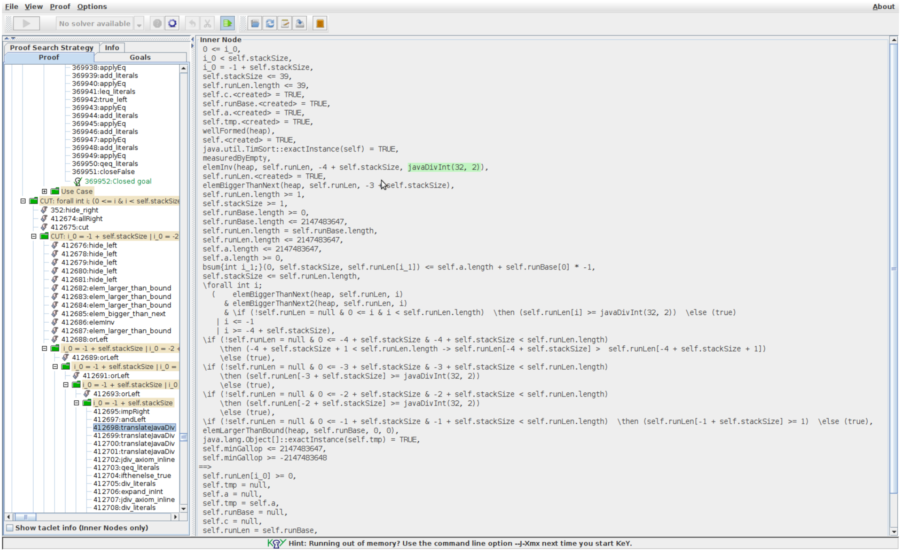
`requires(pushRun) &&` cl. invariant `==> stackSize < len.length`

Proof.

Note first: cl. invariant → `stackSize` ≤ `len.length`.
Assume by contradiction that `stackSize`= `len.length` and do a case distinction on `a.length`. We treat `a.length <= 119`:

1. `len.length` = 4 (from cl. invariant, ln 3).
2. Abbreviate `len[0]+...+len[3] = SUM`, then (`pushRun` ln 7–10) `len[3] >= 16`, `len[2] >= 17`, `len[1] >= 34` and `len[0] >= 52`. Therefore: `SUM >= 16+17+34+52=119`
3. `base[3]` + `len[3]` = `base[0]` + `SUM` (from cl. invariant, ln 11–12)
4. Previous line, with `pushRun` ln 4–5 implies: `runBase + runLen` = `base[0]` + `SUM` + `runLen`
5. But `base[0] >= 0` (cl invariant ln 10) and `runLen>0` (pushrun ln 3), contradicting `runBase + runLen <= 119` (pushRun ln 6)

One proof step in KeY



Evaluation

Proof Stats - summary

	# Rule Apps	# Interactive	LoSpec	LoC
total	2.211.263	5.029	334	333
`pushRun`	26.248	94	17	5
`mergeCollapse`	415.133	1.529	47	13

Evaluation

Proof Stats - summary

	# Rule Apps	# Interactive	LoSpec	LoC
total	2.211.263	5.029	334	333
`pushRun`	26.248	94	17	5
`mergeCollapse`	415.133	1.529	47	13

Evaluation of the problem

- Bug unlikely to be encountered by accident
- Possible security hazard: bug may be exploitable in DoS attack
- Extensive testing unable to expose bug:
  input size too large, structure too complex
- Manual code reviews (Google) unable to expose bug
- Core libraries in widely used languages can contain subtle bugs undetected for years

Responses: general public

- Scientific paper (CAV 2015), articles (ERCIM, Bits & Chips)
- Published blog post viewed 361274 times.

Responses: general public

- Scientific paper (CAV 2015), articles (ERCIM, Bits & Chips)
- Published blog post viewed 361274 times.

Joshua Bloch @joshbloch Follow

Congratulations to Stijn de Gouw et al. for finding and fixing a bug in TimSort using formal methods! envisage-project.eu/proving-androi…

12:33 AM - 25 Feb 2015

126 RETWEETS 65 FAVORITES

JetBrains IntelliJ IDEA Blog (March 10, 2015):
"The first of its kind, this result is both an important development for the Java community and a proof of concept for the feasibility of formal verification and automated theorem proving. Perhaps more importantly, the tool used to detect and identify this bug is completely open source and available to try yourself."

Damian Gryski @dgryski Follow

These #golang timsorts go-search.org/search?q=timso… probably need to be fixed envisage-project.eu/proving-androi… .

5:17 PM - 24 Feb 2015

7 RETWEETS 9 FAVORITES

Messages (8)

msg236533 - (view)	Author: Tim Peters (tim.peters) *	Date: 2015-02-24 19:32

```
Some researchers found an error in the logic of merge_collapse, explained here, and with corrected code shown in section 3.2:

http://envisage-project.eu/proving-android-java-and-python-sorting-algorithm-is-broken-and-how-to-fix-it/

This affects all current versions of Python.  However, I marked the priority "low" because, as the article also notes, there's currently no machine
in existence with enough memory to hold an array large enough for a contrived input to trigger an overflow of the pending-runs stack.

It should be fixed anyway, and their suggested fix looks good to me.
```

Peter O'Hearn
February 24

An impressive use of formal methods. Congrats to the team behind this for their dedicated and valuable work.
http://envisage-project.eu/proving-android-java-and-python…/

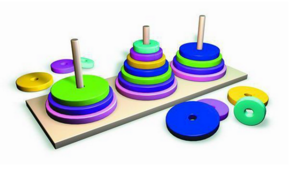
```
mergeCollapse && n = sta
nLen[n-1] > runLen[n] +
nLen[n-2] > runLen[n-1]
runLen[n] > runLen[n+1]
               ==> \ensur
```

Proving that Android's, Java's and Python's sorting algorithm is broken (and showing how to fix...

10.03.2015 08:17 — heise Developer « Vorige | Nächste »

Fehler in Standardsortieralgorithmus mit formalen Methoden aufgedeckt

vorlesen / MP3-Download



Android, Java und Groovy nutzen alle den TimSort-Algorithmus. Informatiker eines Verbundprojekts konnten mit Hilfe eines von ihnen entwickelten Tools

Pascal Hartig @passy 26 Feb
The bug found in the Python and Java implementations of Timsort was also present in this Haskell version: github.com/timjb/haskell-…

Pascal Hartig @passy Follow

I wonder if the invariant could have been encoded in the type system, or if LiquidHaskell, Iris, Agda impls would have exposed this.

4:23 PM - 26 Feb 2015

Responses: developer communities

Java

- Submitted bug report to Java issue tracker

Responses: developer communities

Java

- Submitted bug report to Java issue tracker
- Bug was previously found and "fixed" by increasing `runLen.length`

```
int stackLen = (len <    120  ?  5 :
                len <   1542  ? 10 :
                len < 119151  ? 19 24
                              : 40);
runBase = new int[stackLen];
runLen = new int[stackLen];
```

(In the code, "19" is struck through and replaced by "24".)

Responses: developer communities

Java

- Submitted bug report to Java issue tracker
- Bug was previously found and "fixed" by increasing `runLen.length`
- Bug now fixed by further increasing `runLen.length` based on worst-case analysis

Discussion on OpenJDK mailing list

Stack length increased previously by JDK-8011944 was insufficient for some cases. Please review and push - Lev Priima, 11 Feb 2015

```
int stackLen = (len <    120  ?  5 :
                len <   1542  ? 10 :
                len < 119151  ? 24 :
                               ~~40~~ 49);
runBase = new int[stackLen];
runLen = new int[stackLen];
```

Responses: developer communities

Java

- Submitted bug report to Java issue tracker
- Bug was previously found and "fixed" by increasing `runLen.length`
- Bug now fixed by further increasing `runLen.length` based on worst-case analysis

Discussion on OpenJDK mailing list

Stack length increased previously by JDK-8011944 was insufficient for some cases. Please review and push
- Lev Priima, 11 Feb 2015

Hi Lev, The fix looks fine. Did you consider the improvements suggested in the paper to reestablish the invariant?
- Roger Riggs, Feb 11, 2015

Just briefly looked at it, w/o evaluating formal proof ...
- Lev Priima, Feb 11, 2015

Responses: developer communities

Java

- Submitted bug report to Java issue tracker
- Bug was previously found and "fixed" by increasing `runLen.length`
- Bug now fixed by further increasing `runLen.length` based on worst-case analysis
- Purported class invariant still broken
- Not amenable to mechanic verification

Python

- Bug report filed by Tim Peters
- Bug fixed by checking last 4 runs (verified version)

Android

- No bug report or fix so far

Formal methods work!

Useful links

Blog post

http://tinyurl.com/timsort-bug

Website with full paper, test programs and proofs

http://www.envisage-project.eu/timsort-specification-and-verification

KeY (Java theorem prover)

http://www.key-project.org

Timsort description

http://bugs.python.org/file4451/timsort.txt

OpenJDK dev discussion

http://mail.openjdk.java.net/pipermail/core-libs-dev/2015-February/thread.html#31405