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

CWI Scientific Meeting
Amsterdam, September 25, 2015

http://www.envisage-project.eu
Library

Collection of algorithms with a well-defined interface that perform a commonly used task
Library
Collection of algorithms with a well-defined interface that perform a commonly used task

Example: Java standard library functions

Programming to interfaces:

- Sorting a given array `a`
  ```java
  static void sort(Object[] a)
  ```

- Searching a value `key` in the array `a`
  ```java
  static int binarySearch(Object[] a, Object key)
  ```

Usability of programming language partially depends on good libraries
Library

Collection of algorithms with a well-defined interface that perform a commonly used task

Example: Java standard library functions

Programming to interfaces:

- Sorting a given array `a`
  ```java
  static void sort(Object[] a)
  ```

- Searching a value `key` in the array `a`
  ```java
  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).
**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 (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!

Why analyze Timsort?

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

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

Merging: if (1) is false and $\text{runLen}[n-3] < \text{runLen}[n-1]$, merge runs at $n-3$ and $n-2$, otherwise at $n-2$ and $n-1$

- At the end: merge all runs, resulting in a sorted array

Example, ignoring length $\geq 16$ requirement

<table>
<thead>
<tr>
<th>Input</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>4</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>runLen</th>
</tr>
</thead>
</table>
The algorithm

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

Merging: if (1) is false and $\text{runLen}[n-3] < \text{runLen}[n-1]$, merge runs at $n-3$ and $n-2$, otherwise at $n-2$ and $n-1$

- At the end: merge all runs, resulting in a sorted array

Example, ignoring length $\geq 16$ requirement

<table>
<thead>
<tr>
<th>Input</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>runLen</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The algorithm

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

  Merging: if (1) is false and $\text{runLen}[n-3] < \text{runLen}[n-1]$, merge runs at $n-3$ and $n-2$, otherwise at $n-2$ and $n-1$

- At the end: merge all runs, resulting in a sorted array

Example, ignoring length $\geq 16$ requirement

<table>
<thead>
<tr>
<th>Input</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>runLen</td>
<td>5</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The algorithm

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

Merging: if (1) is false and $\text{runLen}[n-3] < \text{runLen}[n-1]$, merge runs at $n-3$ and $n-2$, otherwise at $n-2$ and $n-1$

- At the end: merge all runs, resulting in a sorted array

Example, ignoring length $\geq 16$ requirement

<table>
<thead>
<tr>
<th>Input</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>4</th>
</tr>
</thead>
</table>

| runLen | 5 | 3 | 2 |
Timsort (II)

The algorithm

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

Merging: if (1) is false and \( \text{runLen}[n-3] < \text{runLen}[n-1] \), merge runs at \( n-3 \) and \( n-2 \), otherwise at \( n-2 \) and \( n-1 \)
- At the end: merge all runs, resulting in a sorted array

Example, ignoring length \( \geq 16 \) requirement

<table>
<thead>
<tr>
<th>Input</th>
<th>1 2 3 4 5 0 1 1 0 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>runLen</td>
<td>5 3 2</td>
</tr>
</tbody>
</table>
The algorithm

- Find next already sorted segment ("runs") extending to length $\geq 16$ with insertion sort.
- Add length of new run to $\text{runLen}$ array
- Merge until last 3 runs satisfy two conditions ("the invariant")
  1. $\text{runLen}[n-3] > \text{runLen}[n-2] + \text{runLen}[n-1]$ (if $n>2$)
  2. $\text{runLen}[n-2] > \text{runLen}[n-1]$ (if $n>1$)
- Merging: if (1) is false and $\text{runLen}[n-3] < \text{runLen}[n-1]$, merge runs at $n-3$ and $n-2$, otherwise at $n-2$ and $n-1$
- At the end: merge all runs, resulting in a sorted array

Example, ignoring length $\geq 16$ requirement

<table>
<thead>
<tr>
<th>Input</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>runLen</td>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Timsort (II)

The algorithm

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

Merging: if (1) is false and \( \text{runLen}[n-3] < \text{runLen}[n-1] \), merge runs at \( n-3 \) and \( n-2 \), otherwise at \( n-2 \) and \( n-1 \)
- At the end: merge all runs, resulting in a sorted array

Example, ignoring length ≥ 16 requirement

<table>
<thead>
<tr>
<th>Input</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>runLen</td>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The algorithm

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

  Merging: if (1) is false and $\text{runLen}[n-3] < \text{runLen}[n-1]$, merge runs at $n-3$ and $n-2$, otherwise at $n-2$ and $n-1$

- At the end: merge all runs, resulting in a sorted array

Example, ignoring length $\geq 16$ requirement

<table>
<thead>
<tr>
<th>Input</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>runLen</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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")
  2. runLen[n-2] > runLen[n-1] (if n>1)

Merging: if (1) is false and runLen[n-3] < runLen[n-1], merge runs at n-3 and n-2, otherwise at n-2 and n-1

- 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
Breaking the invariant

Size of \text{runLen}

1. \text{runLen}[i-2] > \text{runLen}[i-1] + \text{runLen}[i]
2. \text{runLen}[i-1] > \text{runLen}[i]

If the above invariant is true for all \textit{i} and \text{runLen}[i] \geq 16, then

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

Size of `runLen`

1. \( \text{runLen}[i-2] > \text{runLen}[i-1] + \text{runLen}[i] \)
2. \( \text{runLen}[i-1] > \text{runLen}[i] \)

If the above invariant is true for all \( i \) and \( \text{runLen}[i] \geq 16 \), then

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

```java
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[i-2] > runLen[i-1] + runLen[i]
2. runLen[i-1] > runLen[i]

If the above invariant is true for all \( i \) and \( \text{runLen}[i] \geq 16 \), 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 \( \text{runLen} \)

1. \( \text{runLen}[i-2] > \text{runLen}[i-1] + \text{runLen}[i] \)
2. \( \text{runLen}[i-1] > \text{runLen}[i] \)

If the above invariant is true for all \( i \) and \( \text{runLen}[i] \geq 16 \), 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 \( \text{runLen}[i-2] < \text{runLen}[i] \): merge at idx \( i-2 \) and \( i-1 \), otherwise merge runs at indices \( i-1 \) and \( i \)

\[
\begin{array}{cccc}
120 & 80 & 25 & 20 \\
\end{array}
\]
Breaking the invariant

Size of \( \text{runLen} \)

1. \( \text{runLen}[i-2] > \text{runLen}[i-1] + \text{runLen}[i] \)
2. \( \text{runLen}[i-1] > \text{runLen}[i] \)

If the above invariant is true for all \( i \) and \( \text{runLen}[i] \geq 16 \), 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 \( \text{runLen}[i-2] < \text{runLen}[i] \): merge at idx \( i-2 \) and \( i-1 \), otherwise merge runs at indices \( i-1 \) and \( i \)

| runLen | 120 | 80 | 25 | 20 | 30 |
Breaking the invariant

### Size of runLen

1. \( \text{runLen}[i-2] > \text{runLen}[i-1] + \text{runLen}[i] \)
2. \( \text{runLen}[i-1] > \text{runLen}[i] \)

If the above invariant is true for all \( i \) and \( \text{runLen}[i] \geq 16 \), 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 \( \text{runLen}[i-2] < \text{runLen}[i] \): merge at idx \( i-2 \) and \( i-1 \), otherwise merge runs at indices \( i-1 \) and \( i \)

| runLen | 120 | 80 | 25 | 20 | 30 |
Breaking the invariant

Size of runLen

1. \( \text{runLen}[i-2] > \text{runLen}[i-1] + \text{runLen}[i] \)
2. \( \text{runLen}[i-1] > \text{runLen}[i] \)

If the above invariant is true for all \( i \) and \( \text{runLen}[i] \geq 16 \), 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 \( \text{runLen}[i-2] < \text{runLen}[i] \): merge at idx \( i-2 \) and \( i-1 \), otherwise merge runs at indices \( i-1 \) and \( i \)

runLen | 120 | 80 | 45 | 30
Breaking the invariant

Size of \textit{runLen}

1. \texttt{runLen[i-2] > runLen[i-1] + runLen[i]}
2. \texttt{runLen[i-1] > runLen[i]}

If the above invariant is true for all \textit{i} and \texttt{runLen[i] >= 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 \texttt{runLen[i-2] < runLen[i]}: merge at idx \texttt{i-2 and i-1}, otherwise merge runs at indices \texttt{i-1 and i}

| 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

<table>
<thead>
<tr>
<th>Language</th>
<th>Smallest array that triggers error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Android</td>
<td>65.536 ( (2^{16}) )</td>
</tr>
<tr>
<td>Java</td>
<td>67.108.864 ( (2^{26}) )</td>
</tr>
<tr>
<td>Python</td>
<td>562.949.953.421.312 ( (2^{49}) )</td>
</tr>
</tbody>
</table>

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 \( \text{runLen} \) to store run lengths

<table>
<thead>
<tr>
<th>Language</th>
<th>Smallest array that triggers error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Android</td>
<td>65.536 ( (2^{16}) )</td>
</tr>
<tr>
<td>Java</td>
<td>67.108.864 ( (2^{26}) )</td>
</tr>
<tr>
<td>Python</td>
<td>562.949.953.421.312 ( (2^{49}) )</td>
</tr>
</tbody>
</table>

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

Provided worst-case analysis of broken version

- Shows the actual minimally required \( \text{runLen.length} \)
Our work (II)

Fixed the algorithm

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

```java
/** ... 
 * 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);
    }
}
```
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
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

- Deductive Java theorem prover developed by TU Darmstadt (Reiner Haehnle), Karlsruhe, Chalmers
- termination, absence of the bug, and all other run-time exceptions
- this requires: formal specifications for all functions
Proof Stats - summary

<table>
<thead>
<tr>
<th></th>
<th># Rule Apps</th>
<th># Interactive</th>
<th>LoSpec</th>
<th>LoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>total</td>
<td>2,211,263</td>
<td>5,029</td>
<td>334</td>
<td>333</td>
</tr>
</tbody>
</table>

Verification effort: \(\approx 2.5\) man months
Evaluation

Proof Stats - summary

<table>
<thead>
<tr>
<th></th>
<th># Rule Apps</th>
<th># Interactive</th>
<th>LoSpec</th>
<th>LoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>total</td>
<td>2,211,263</td>
<td>5,029</td>
<td>334</td>
<td>333</td>
</tr>
</tbody>
</table>

Verification effort: \(\approx 2.5\) man months

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
- Not amenable to model-checking
Responses: developer communities

Java

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

```java
int stackLen = (len < 120 ? 5 :
                len < 1542 ? 10 :
                len < 119151 ? 19 : 40);
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

```java
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 assertional, 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
Popular articles (ERCIM, Bits & Chips)
Published blog post viewed 1441160 times.
Responses: general public

- Popular articles (ERCIM, Bits & Chips)
- Published blog post viewed 1441160 times.
Formal methods work!
<table>
<thead>
<tr>
<th>Useful links</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Blog post</strong></td>
</tr>
<tr>
<td><a href="http://tinyurl.com/timsort-bug">http://tinyurl.com/timsort-bug</a></td>
</tr>
<tr>
<td><strong>Website with full paper, test programs and proofs</strong></td>
</tr>
<tr>
<td><strong>KeY (Java theorem prover)</strong></td>
</tr>
<tr>
<td><a href="http://www.key-project.org">http://www.key-project.org</a></td>
</tr>
<tr>
<td><strong>Timsort description</strong></td>
</tr>
<tr>
<td><a href="http://bugs.python.org/file4451/timsort.txt">http://bugs.python.org/file4451/timsort.txt</a></td>
</tr>
<tr>
<td><strong>OpenJDK dev discussion</strong></td>
</tr>
<tr>
<td><a href="http://mail.openjdk.java.net/pipermail/core-libs-dev/2015-February/thread.html#31405">http://mail.openjdk.java.net/pipermail/core-libs-dev/2015-February/thread.html#31405</a></td>
</tr>
</tbody>
</table>