A Brief History of TimSort

Our contributions are written in blue.

invented by Tim Peters
sorting algorithm in Python
used in Android
Java
Octave

old TimSort

bug uncovered! [1]
Python ➔ certified fix
Java ➔ custom fix

new bug in Java! [3]
refined analysis: $\mathcal{O}(n + nH)$

2001 '02 '03 '04 '05 '06 '07 '08 '09 '10 '11 '12 '13 '14 '15 '16 '17 '18 '19 '20 '21

TimSort Principle

Natural decomposition of a sequence into maximal monotonic runs:

1, 2, 3, 4, ... are the top-most runs on the stack. The length of the $i^{th}$ run is denoted by $r_i$.

1, 2, 3, 4, 7, 24, 28, 50, 6, 4, 20, 8, 1:
(stack values represent run lengths)

Algorithm

get a run decomposition of the initial sequence
start with an empty stack
while the run decomposition is not empty do
| take the next run and push it onto the stack
| if $r_1 > r_2$
| then new 1
| merge 1 & 2
| else if
| if $r_1 > r_2$
| then new 1
| merge 1 & 2
| or
| if $r_1 + r_2 > r_3$
| then new 2
| merge 2 & 3
| or
| if $r_1 + r_2 > r_1$
| then new 3
| merge 3 & 4

Example of stack evolution for runs of length 24, 18, 50, 28, 20, 6, 4, 8, 1:
(stack values represent run lengths)

Results

13 years after its announcement by Tim Peters, we obtained the first proof of its worst-case running time:

Theorem [2, 3]: TimSort runs in $\mathcal{O}(n \log n)$ time.

By design TimSort is well suited for partially sorted data. It falls into the adaptive sort family. Taking the number of runs $\rho$ as a (natural) parameter for a refined analysis we obtained:

Theorem [3]: TimSort runs in $\mathcal{O}(n + nH)$ time.

Going further, we introduce the notion of entropy $H$ to take the variations in run lengths into account, defined by:

$H = -\sum \frac{r_i}{n} \log_2 \left( \frac{r_i}{n} \right)$

Theorem [3]: TimSort runs in $\mathcal{O}(n + nH)$ time.

References


Our in-depth analysis of the algorithm enabled us to produce a bug in Java sorting! It was fixed within a few weeks after we reported it.