Timsort

Timsort is a hybrid stable sorting algorithm, derived from merge sort and insertion sort, designed to perform well on many kinds of real-world data. It was implemented by Tim Peters in 2002 for use in the Python programming language. The algorithm finds subsequences of the data that are already ordered (runs) and uses them to sort the remainder more efficiently. This is done by merging runs until certain criteria are fulfilled. Timsort has been Python's standard sorting algorithm since version 2.3. It is also used to sort arrays of non-primitive type in Java SE 7,[4] on the Android platform,[5] in GNU Octave,[6] Google Chrome,[7] and Swift[8].

It uses techniques from Peter McIlroy's 1993 paper "Optimistic Sorting and Information Theoretic Complexity".[9]

Timsort

Class	Sorting algorithm
Data structure	Array
Worst-case performance	$O(n \log n)$ [1][2]
Best-case performance	$O(n)$[3]
Average performance	$O(n \log n)$
Worst-case space complexity	$O(n)$

Contents

Operation

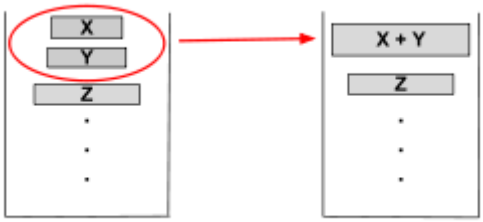
Timsort was designed to take advantage of *runs* of consecutive ordered elements that already exist in most real-world data, *natural runs*. It iterates over the data collecting elements into runs and simultaneously putting those runs in a stack. Whenever the runs on the top of the stack match a merge criteria, they are merged together. This goes on until all data is traversed; then, all runs are merged two at a time and only one sorted run remains. The advantage of merging ordered runs instead of merging fixed size sub-lists (as done by traditional mergesort) is that it decreases the total number of comparisons needed to sort the entire list.

Each run has a minimum size, which is based on the size of the input and it is defined at the start of the algorithm. If a run is smaller than this minimum run size, insertion sort is used to add more elements to the run until the minimum run size is reached.

Merge criteria

Timsort is a stable sorting algorithm (order of elements with same key is kept) and strives to perform balanced merges (merging merges runs of similar sizes).

In order to achieve sorting stability, only consecutive runs are merged. Between non-consecutive two runs, there can be an element with the same key of elements inside the runs and merging those two runs would change the order of equal keys. Example of this situation ([] are ordered runs): [1 2 2] 1 4 2 [0 1 2]



The runs are inserted in a stack. If $|Z| \leq |Y| + |X|$, then X and Y are merged and replaced on the stack. In this way, merging is continued until all runs satisfy i. $|Z| > |Y| + |X|$ and ii. $|Y| > |X|$.

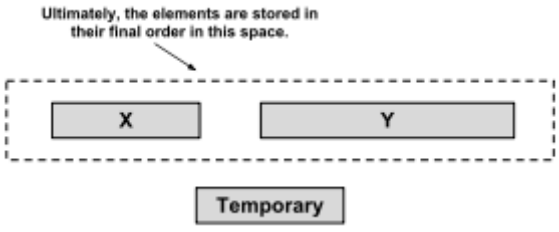
In pursuit of balanced merges, Timsort considers three runs on the top of the stack, X, Y, Z, and maintains the invariants:

i. $|Z| > |Y| + |X|$
ii. $|Y| > |X|$[10]

If any of these invariants are violated, Y is merged with the smaller of X or Z and the invariants are checked again. Once the invariants hold, the search for a new run in the data can start.[11] These invariants maintain merges as being approximately balanced while maintaining a compromise between delaying merging for balance, exploiting fresh occurrence of runs in cache memory and making merge decisions relatively simple.

Merge space overhead

The original merge sort implementation is not in-place and it has an space overhead of N (data size). in-place merge sort implementations exist, but have a high time overhead. In order to achieve a middle term, Timsort performs a merge sort with a small time overhead and smaller space overhead than N.



To merge, Timsort copies the elements of the smaller array (X in this illustration) to temporary memory, then sorts and fills elements in final order into the combined space of X and Y.

First, Timsort performs a binary search to find the location where the first element of the second run would be inserted in the first ordered run, keeping it ordered. Then, it performs the same algorithm to find the location where the last element of the first run would be inserted in the second ordered run, keeping it ordered. Elements before and after these locations are already in their correct place and do not need to be merged. Then, the smaller of the remaining elements of the two runs is copied into temporary memory, and elements are merged with the larger run into the now free space. If the first run is smaller, the merge

starts at the beginning; if the second is smaller, the merge starts at the end. This optimization reduces the number of required element movements, the running time and the temporary space overhead in the general case.

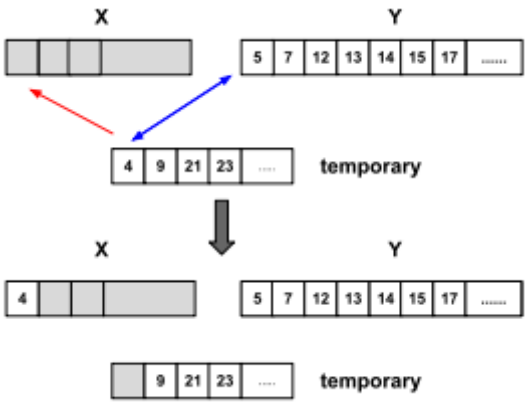
Example: two runs [1, 2, 3, 6, 10] and [4, 5, 7, 9, 12, 14, 17] must be merged. Note that both runs are already sorted individually. The smallest element of the second run is 4 and it would have to be added at the 4th position of the first run in order to preserve its order (assuming that the first position of a run is 1). The largest element of the first run is 10 and it would have to be added at the 5th position of the second run in order to preserve its order. Therefore, [1, 2, 3] and [12, 14, 17] are already in their final positions and the runs in which elements movements are required are [6, 10] and [4, 5, 7, 9]. With this knowledge, we only need to allocate a temporary buffer of size 2 instead of 5.

Merge direction

Merging can be done in both directions: left-to-right, as in the traditional mergesort, or right-to-left.

Galloping mode during merge

An individual merge of runs R1 and R2 keeps the count of consecutive elements selected from a run. When this number reaches the *minimum galloping threshold* (*min_gallop*), Timsort considers that it is likely that many consecutive elements may still be selected from that run and switches to the galloping mode. Let us assume that R1 is the responsible for triggering it. In this mode, the algorithm performs an exponential search, also known as galloping search, for the next element x of the run R2 in the run R1. This is done in two stages: the first one finds the range ($2^k - 1$, $2^{k+1} - 1$) where x is. The second stage performs a binary search for the element x in the range found in the first stage. The galloping mode is an attempt to adapt the merge algorithm to the pattern of intervals between elements in runs.



Elements (pointed to by blue arrow) are compared and the smaller element is moved to its final position (pointed to by red arrow).

Galloping is not always efficient. In some cases galloping mode requires more comparisons than a simple linear search. According to benchmarks done by the developer, galloping is beneficial only when the initial element of one run is not one of the first seven elements of the other run. This implies an initial threshold of 7. To avoid the drawbacks of galloping mode, two actions are taken: (1) When galloping is found to be less efficient than binary search, galloping mode is exited. (2) The success or failure of galloping is used to adjust *min_gallop*. If the selected element is from the same array that returned an element previously, *min_gallop* is reduced by one, thus encouraging the return to galloping mode. Otherwise, the value is incremented by one, thus discouraging a return to galloping mode. In the case of random data, the value of *min_gallop* becomes so large that galloping mode never recurs.[12]

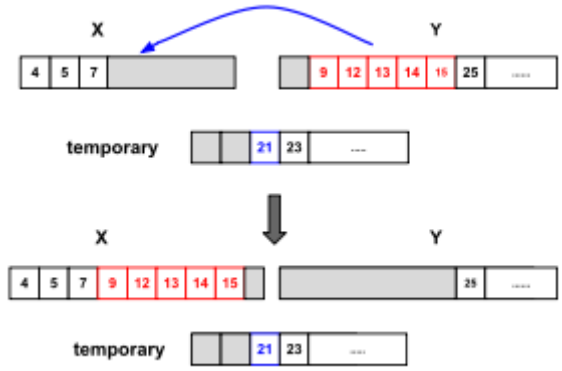
Descending runs

In order to also take advantage of data sorted in descending order, Timsort inverses strictly descending runs when it finds them and add them to the stack of runs. Since descending runs are later blindly reversed, excluding runs with equal elements maintains the algorithm's stability; i.e., equal elements won't be reversed.

Minimum run size

Because merging is most efficient when the number of runs is equal to, or slightly less than, a power of two, and notably less efficient when the number of runs is slightly more than a power of two, Timsort chooses *minrun* to try to ensure the former condition.[10]

Minrun is chosen from the range 32 to 64 inclusive, such that the size of the data, divided by *minrun*, is equal to, or slightly less than, a power of two. The final algorithm takes the six most significant bits of the size of the array, adds one if any of the remaining bits are set, and uses that result as the *minrun*. This algorithm works for all arrays, including those smaller than 64; for arrays of size 63 or less, this sets *minrun* equal to the array size and Timsort reduces to an insertion sort.[10]



All red elements are smaller than blue (here, 21). Thus they can be moved in a chunk to the final array.

Timsort algorithm searches for minimum-size ordered sequences, minruns, to perform its sort.

Analysis

In the worst case, Timsort takes $O(n \log n)$ comparisons to sort an array of n elements. In the best case, which occurs when the input is already sorted, it runs in linear time, meaning that it is an adaptive sorting algorithm.[3]

It is advantageous over Quicksort for sorting object references or pointers because these require expensive memory indirection to access data and perform comparisons and Quicksort's cache coherence benefits are greatly reduced.

Formal verification

In 2015, Dutch and German researchers in the EU FP7 ENVISAGE project found a bug in the standard implementation of Timsort.[13]

Specifically, the invariants on stacked run sizes ensure a tight upper bound on the maximum size of the required stack. The implementation preallocated a stack sufficient to sort 2^{64} bytes of input, and avoided further overflow checks.

However, the guarantee requires the invariants to apply to *every* group of three consecutive runs, but the implementation only checked it for the top three.[13] Using the KeY tool for formal verification of Java software, the researchers found that this check is not sufficient, and they were able to find run lengths

(and inputs which generated those run lengths) which would result in the invariants being violated deeper in the stack after the top of the stack was merged.[14]

As a consequence, for certain inputs the allocated size is not sufficient to hold all unmerged runs. In Java, this generates for those inputs an array-out-of-bound exception. The smallest input that triggers this exception in Java and Android v7 is of size 67 108 864 (2^{26}). (Older Android versions already triggered this exception for certain inputs of size 65 536 (2^{16}))

The Java implementation was corrected by increasing the size of the preallocated stack based on an updated worst-case analysis. The article also showed by formal methods how to establish the intended invariant by checking that the *four* topmost runs in the stack satisfy the two rules above. This approach was adopted by Python[15] and Android.

References

1. Peters, Tim. "[Python-Dev] Sorting" (http://mail.python.org/pipermail/python-dev/2002-July/026837.html). *Python Developers Mailinglist*. Retrieved 24 February 2011. "[Timsort] also has good aspects: It's stable (items that compare equal retain their relative order, so, e.g., if you sort first on zip code, and a second time on name, people with the same name still appear in order of increasing zip code; this is important in apps that, e.g., refine the results of queries based on user input). ... It has no bad cases (O(N log N) is worst case; N−1 compares is best)."
2. "[DROPS]" (http://drops.dagstuhl.de/opus/volltexte/2018/9467/). Retrieved 1 September 2018. "TimSort is an intriguing sorting algorithm designed in 2002 for Python, whose worst-case complexity was announced, but not proved until our recent preprint."
3. Chandramouli, Badrish; Goldstein, Jonathan (2014). *Patience is a Virtue: Revisiting Merge and Sort on Modern Processors*. SIGMOD/PODS.
4. "[#JDK-6804124] (coll) Replace "modified mergesort" in java.util.Arrays.sort with timsort" (https://bugs.openjdk.java.net/browse/JDK-6804124). *JDK Bug System*. Retrieved 11 June 2014.
5. "Class: java.util.TimSort<T>" (https://web.archive.org/web/20150716000631/https://android.googlesource.com/platform/libcore/+/gingerbread/luni/src/main/java/java/util/TimSort.java). *Android Gingerbread Documentation*. Archived from the original (https://android.googlesource.com/platform/libcore/+/gingerbread/luni/src/main/java/java/util/TimSort.java) on 16 July 2015. Retrieved 24 February 2011.
6. "liboctave/util/oct-sort.cc" (http://hg.savannah.gnu.org/hgweb/octave/file/0486a29d780f/liboctave/util/oct-sort.cc). *Mercurial repository of Octave source code*. Lines 23-25 of the initial comment block. Retrieved 18 February 2013. "Code stolen in large part from Python's, listobject.c, which itself had no license header. However, thanks to Tim Peters for the parts of the code I ripped-off."
7. "Getting things sorted in V8 · V8" (https://v8.dev/blog/array-sort). *v8.dev*. Retrieved 21 December 2018.
8. "Is sort() stable in Swift 5?" (https://forums.swift.org/t/is-sort-stable-in-swift-5/21297/9). *Swift Forums*. 4 July 2019. Retrieved 4 July 2019.
9. McIlroy, Peter (January 1993). "Optimistic Sorting and Information Theoretic Complexity". *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*. pp. 467–474. ISBN 0-89871-313-7.
10. "listsort.txt" (https://hg.python.org/cpython/file/tip/Objects/listsort.txt). *Python source code*. 10 February 2017.

11. MacIver, David R. (11 January 2010). "Understanding timsort, Part 1: Adaptive Mergesort" (http://www.drmaciver.com/2010/01/understanding-timsort-1adaptive-mergesort/). Retrieved 5 December 2015.
12. Peters, Tim. "listsort.txt" (https://github.com/python/cpython/blob/master/Objects/listsort.txt). *CPython git repository*. Retrieved 5 December 2019.
13. de Gouw, Stijn; Rot, Jurriaan; de Boer, Frank S.; Bubel, Richard; Hähnle, Reiner (July 2015). "OpenJDK's Java.utils.Collection.sort() Is Broken: The Good, the Bad and the Worst Case" (http://envisage-project.eu/wp-content/uploads/2015/02/sorting.pdf) (PDF). *Computer Aided Verification*: 273–289. doi:10.1007/978-3-319-21690-4_16 (https://doi.org/10.1007%2F978-3-319-21690-4_16).
14. de Gouw, Stijn (24 February 2015). "Proving that Android's, Java's and Python's sorting algorithm is broken (and showing how to fix it)" (http://envisage-project.eu/proving-android-java-and-python-sorting-algorithm-is-broken-and-how-to-fix-it/). Retrieved 6 May 2017.
15. Python Issue Tracker – Issue 23515: Bad logic in timsort's merge_collapse (http://bugs.python.org/issue23515)

Further reading

- Auger, Nicolas; Nicaud, Cyril; Pivoteau, Carine (2015). "Merge Strategies: from Merge Sort to TimSort" (https://hal-upec-upem.archives-ouvertes.fr/hal-01212839). *hal-01212839*.

External links

- timsort.txt (http://bugs.python.org/file4451/timsort.txt) – original explanation by Tim Peters revised branch version (https://hg.python.org/cpython/file/tip/Objects/listsort.txt)
  - listobject.c:1910@7b5057b89a56 (https://hg.python.org/cpython/file/7b5057b89a56/Objects/listobject.c#l1910) – Python's Tree implementation
- Python's listobject.c (https://hg.python.org/cpython/file/default/Objects/listobject.c) – the C implementation of Timsort used in CPython
- OpenJDK's TimSort.java (http://cr.openjdk.java.net/~martin/webrevs/openjdk7/timsort/raw_files/new/src/share/classes/java/util/TimSort.java) – the Java implementation of Timsort
- GNU Octave's oct-sort.cc (http://hg.savannah.gnu.org/hgweb/octave/file/0486a29d780f/liboctave/util/oct-sort.cc) – the C++ implementation of Timsort used in GNU Octave
- Sort Comparison (http://stromberg.dnsalias.org/~strombrg/sort-comparison/) – a pure Python and Cython implementation of Timsort, among other sorts
- Gee.TimSort (https://git.gnome.org/browse/libgee/tree/gee/timsort.vala) - Vala implementation of Timsort

---