WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
Deutsch
فارسی
Italiano
Português
Русский
Српски / srpski
Edit links

Timsort

From Wikipedia, the free encyclopedia

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 invented by Tim Peters in 2002 for use in the Python programming language. The algorithm finds subsets of the data that are already ordered, and uses that knowledge to sort the remainder more efficiently. This is done by merging an identified subset, called a run, with existing 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,[2] on the Android platform,[3] and in GNU Octave.[4]

Timsort

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

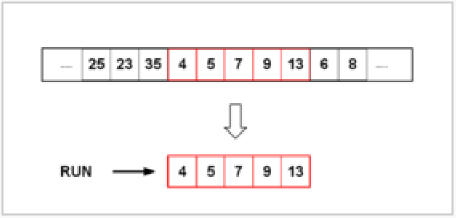
Contents [hide]

Operation [edit]

Timsort was designed to take advantage of partial orderings that already exist in most real-world data. Timsort operates by finding runs, subsets of at least two elements, in the data. Runs are either non-descending (each element is equal to or greater than its predecessor) or strictly descending (each element is lower than its predecessor). If it is descending, it must be strictly descending, since descending runs are later reversed by a simple swap of elements from both ends converging in the middle. After obtaining such a run in the given array, Timsort processes it, and then searches for the next run.

Minrun [edit]

A natural run is a sub-array that is already ordered. Natural runs in real-world data may be of varied lengths. Timsort chooses a sorting technique depending on the length of the run. For example, if the run length is smaller than a certain value, insertion sort is used. Thus Timsort is an adaptive sort.[5]



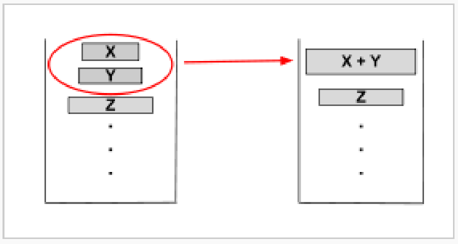
Timsort algorithm searches for such ordered sequences, minruns, to perform its sort

The size of the run is checked against the minimum run size. The minimum run size (minrun) depends on the size of the array. For an array of fewer than 64 elements, minrun is the size of the array, reducing Timsort to an insertion sort. For larger arrays, minrun is chosen from the range 32 to 64 inclusive, such that the size of the array, divided by minrun, is equal to, or slightly smaller 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.[5]

Insertion sort [edit]

When an array is random, natural runs most likely contain fewer than minrun elements. In this case, an appropriate number of succeeding elements is selected, and an insertion sort increases the size of the run to minrun size. Thus, most runs in a random array are, or become, minrun in length. This results in efficient, balanced merges. It also results in a reasonable number of function calls in the implementation of the sort.[6]

Merge memory [edit]



The minruns are inserted in a stack. If X < Y + Z then X and Y are merged and then inserted into a stack. In this way, merging is continued until all arrays satisfy a) X > Y + Z and b) Y > Z

Once run lengths are optimized, the runs are merged. When a run is found, the algorithm pushes its base address and length on a stack. A function determines whether the run should be merged with previous runs. Timsort does not merge non-consecutive runs, because doing this would cause the element common to all three runs to become out of order with respect to the middle run.

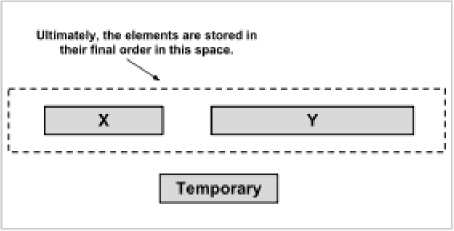
Thus, merging is always done on consecutive runs. For this, the three top-most runs in the stack which are unsorted are considered. If, say, X, Y, Z represent the lengths of the three uppermost runs in the stack, the algorithm merges the runs so that ultimately the following two rules are satisfied:

i. X > Y + Z
ii. Y > Z[5]

For example, if the first of the two rules is not satisfied by the current run status, that is, if X < Y + Z, then, Y is merged with the smaller of X and Z. The merging continues until both rules are satisfied. Then the algorithm determines the next run.[6]

The rules above aim at maintaining run lengths as close to each other as possible to balance the merges. Only a small number of runs are remembered, as the stack is of a specific size. The algorithm exploits the fresh occurrence of the runs to be merged, in cache memory. Thus a compromise is attained between delaying merging, and exploiting fresh occurrence in cache.

Merging procedure [edit]


Ultimately, the elements are stored in their final order in this space.
X
Y
Temporary

Algorithm creates a temporary memory equal to size of smaller array. Then, it shifts elements in (say if X is smaller) X to the temporary memory and then sorts and fills elements in final order into combined space of X and Y

Merging adjacent runs is done with the help of temporary memory. The temporary memory is of the size of the lesser of the two runs. The algorithm copies the smaller of the two runs into this temporary memory and then uses the original memory (of the smaller run) and the memory of the other run to store sorted output.

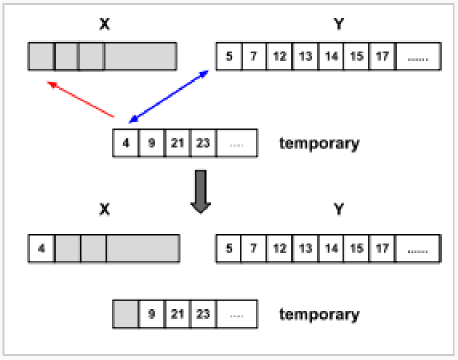
A simple merge algorithm runs left to right or right to left depending on which run is smaller, on the temporary memory and original memory of the larger run. The final sorted run is stored in the original memory of the two initial runs. Timsort searches for appropriate positions for the starting element of one array in the other using an adaptation of binary search.

Say, for example, two runs A and B are to be merged, with A as the smaller run. In this case a binary search examines A to find the first position larger than the first element of B (a'). Note that A and B are already sorted individually. When a' is found, the algorithm can ignore elements before that position while inserting B. Similarly, the algorithm also looks for the smallest element in B (b') greater than the largest element in A (a"). The elements after b' can also be ignored for the merging. This preliminary searching is not efficient for highly random data, but is efficient in other situations and is hence included.

Galloping mode [edit]

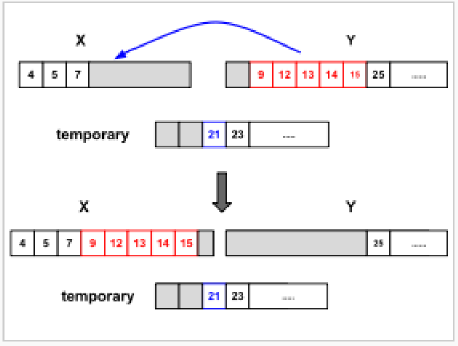
Most of the merge occurs in what is called "one pair at a time" mode, where respective elements of both runs are compared. When the algorithm merges left-to-right, the smaller of the two is brought to a merge area. A count of the number of times the final element appears in a given run is recorded. When this value reaches a certain threshold, MIN_GALLOP, the merge switches to "galloping mode". In this mode we use the previously mentioned adaptation of binary search to identify where the first element of the smaller array must be placed in the larger array (and vice versa). All elements in the larger array that occur before this location can be moved to the merge area as a group (and vice versa). The functions *merge-lo* and *merge-hi* increment the value of min-gallop (initialized to MIN_GALLOP), if galloping is not efficient, and decrement it if it is. If too many consecutive elements come from different runs, galloping mode is exited.[5]

In galloping mode, the algorithm searches for the first element of one array in the other. This is done by comparing that first element (initial element) with the zeroth element of the other array, then the first, the third and so on, that is $(2^k - 1)$th element, so as to get a range of elements between which the initial element will lie. This shortens the range for binary searching, thus increasing efficiency. Galloping proves to be more efficient except in cases with especially long runs, but random data usually has shorter runs. Also, in cases where galloping is found to be less efficient than binary search, galloping mode is exited.



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. One reason is due to excessive function calls. Function calls are expensive and thus when frequent, they affect program efficiency. In some cases galloping mode requires more comparisons than a simple linear search (one at a time search). While for the first few cases both modes may require the same number of comparisons, over time galloping mode requires 33% more comparisons than linear search to arrive at the same results. Moreover all comparisons in galloping mode are done by function calls.



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

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 a MIN_GALLOP of 7. To avoid the drawbacks of galloping mode, the merging functions adjust the value of min-gallop. If the element is from the array currently that has been returning elements, min-gallop is reduced by one. Otherwise, the value is incremented by one, thus discouraging a return to galloping mode. When this is done, in the case of random data, the value of min-gallop becomes so large that galloping mode never recurs.

In the case where merge-hi is used (that is, merging is done right-to-left), galloping starts from the right end of the data, that is, the last element. Galloping from the beginning also gives the required results, but makes more comparisons. Thus, the galloping algorithm uses a variable that gives the index at which galloping should begin. Timsort can enter galloping mode at any index and continue checking at the next index which is offset by 1, 3, 7,...., $(2^k - 1)$.. and so on from the current index. In the case of merge-hi, the offsets to the index will be -1, -3, -7,....[5]

Performance [edit]

According to information theory, no comparison sort can perform better than $\Theta(n \log n)$ comparisons in the worst case. On real-world data, Timsort often requires far fewer than $\Theta(n \log n)$ comparisons, because it takes advantage of the fact that sublists of the data may already be ordered.[7]

The following table compares the time complexity of Timsort with other comparison sorts.

	Timsort	Introsort	Merge sort	Quicksort	Insertion sort	Selection sort	Smoothsort
Best case	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n)$
Average case	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n \log n)$
Worst case	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n \log n)$

The following table provides a comparison of the space complexities of the various sorting techniques. Note that for merge sort, the *worst case* space complexity is usually $O(n)$, depending on the merge algorithm used. Merge algorithms with $O(1)$ space complexity exist.

	Timsort	Merge sort	Quicksort	Insertion sort	Selection sort	Smoothsort

Space complexity	$O(n)$	$O(n)$	$O(\log n)$	$O(1)$	$O(1)$	$O(1)$

Note, however, that the space complexity of both Timsort and merge sort can be reduced to $\log n$ at the cost of speed (see in-place merge sort).

Debugging with formal methods [edit]

Researchers discovered using formal verification (KeY) that the three runs mentioned above are not sufficient to hold the invariant for any arbitrary array.[8] The bug was not deemed critical because no current machine could hold a sufficient number of elements, approximately 2^{49} or 562 trillion, to trigger the error. The bug was patched in Python a day later.[9]

References [edit]

1. ^ Peters, Tim. "[Python-Dev] Sorting". *Python Developers Mailinglist*. Retrieved 24 Feb 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. ^ "[#JDK-6804124] (coll) Replace "modified mergesort" in java.util.Arrays.sort with timsort". *JDK Bug System*. Retrieved 11 Jun 2014.
3. ^ "Class: java.util.TimSort<T>". *Android Gingerbread Documentation*. Retrieved 24 Feb 2011.[*dead link*]
4. ^ "liboctave/util/oct-sort.cc". *Mercurial repository of Octave source code*. Lines 23-25 of the initial comment block. Retrieved 18 Feb 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."
5. ^ *a b c d e* timsort, python. "python_timsort".
6. ^ *a b* timsort, understanding. "understanding timsort".
7. ^ Martelli, Alex (2006). *Python in a Nutshell (In a Nutshell (O'Reilly))*. O'Reilly Media, Inc. p. 57. ISBN 0-596-10046-9.
8. ^ Proving that Android's, Java's and Python's sorting algorithm is broken (and showing how to fix it)
9. ^ Python Issue Tracker - Issue 23515: Bad logic in timsort's merge_collapse

External links [edit]

- timsort.txt - original explanation by Tim Peters.
- Visualising Timsort - the source for the image on this page.
- Python's listobject.c - the C implementation of timsort for CPython.
- OpenJDK's TimSort.java - the Java implementation of timsort.
- GNU Octave's oct-sort.cc - the C++ implementation of timsort for GNU Octave.
- Sort Comparison - a Pure Python and Cython implementation of Timsort, among other sorts.

v · t · e	**Sorting algorithms** [hide]
Theory	Computational complexity theory · Big O notation · Total order · Lists · Inplacement · Stability · Comparison sort · Adaptive sort · Sorting network · Integer sorting
Exchange sorts	Bubble sort · Cocktail sort · Odd–even sort · Comb sort · Gnome sort · Quicksort · Stooge sort · Bogosort
Selection sorts	Selection sort · Heapsort · Smoothsort · Cartesian tree sort · Tournament sort · Cycle sort
Insertion sorts	Insertion sort · Shellsort · Splaysort · Tree sort · Library sort · Patience sorting
Merge sorts	Merge sort · Cascade merge sort · Oscillating merge sort · Polyphase merge sort · Strand sort
Distribution sorts	American flag sort · Bead sort · Bucket sort · Burstsort · Counting sort · Pigeonhole sort · Proxmap sort · Radix sort · Flashsort
Concurrent sorts	Bitonic sorter · Batcher odd–even mergesort · Pairwise sorting network
Hybrid sorts	Block sort · **Timsort** · Introsort · Spreadsort · JSort
Other	Topological sorting · Pancake sorting · Spaghetti sort

Categories: Sorting algorithms | Comparison sorts | Stable sorts