

# Merge sort

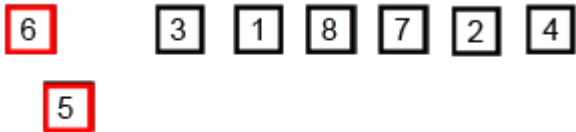
From Wikipedia, the free encyclopedia

In computer science, **merge sort** (also commonly spelled **mergesort**) is an efficient, general-purpose, comparison-based sorting algorithm. Most implementations produce a stable sort, which means that the implementation preserves the input order of equal elements in the sorted output. Mergesort is a divide and conquer algorithm that was invented by John von Neumann in 1945.<sup>[1]</sup> A detailed description and analysis of bottom-up mergesort appeared in a report by Goldstine and Neumann as early as 1948.<sup>[2]</sup>

## Contents

- 1 Algorithm
  - 1.1 Top-down implementation
  - 1.2 Bottom-up implementation
  - 1.3 Top-down implementation using lists
  - 1.4 Bottom-up implementation using lists
- 2 Natural merge sort
- 3 Analysis
- 4 Variants
- 5 Use with tape drives
- 6 Optimizing merge sort
- 7 Parallel merge sort
- 8 Comparison with other sort algorithms
- 9 Notes
- 10 References
- 11 External links

## Merge sort



An example of merge sort. First divide the list into the smallest unit (1 element), then compare each element with the adjacent list to sort and merge the two adjacent lists. Finally all the elements are sorted and merged.

|                             |  |
|-----------------------------|--|
| Class                       | Sorting algorithm                                |
| Data structure              | Array  |
| Worst-case performance      | $O(n \log n)$                                    |
| Best-case performance       | $O(n \log n)$ typical,<br>$O(n)$ natural variant |
| Average performance         | $O(n \log n)$                                    |
| Worst-case space complexity | $O(n)$ total, $O(n)$ auxiliary                   |

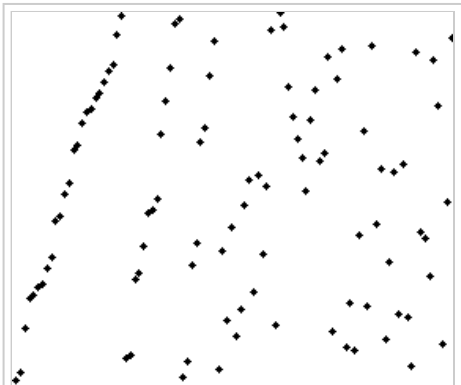
## Algorithm

Conceptually, a merge sort works as follows:

1. Divide the unsorted list into  $n$  sublists, each containing 1 element (a list of 1 element is considered sorted).
2. Repeatedly merge sublists to produce new sorted sublists until there is only 1 sublist remaining. This will be the sorted list.

## Top-down implementation

Example C-like code using indices for top down merge sort algorithm that recursively splits the list (called *runs* in this example) into sublists until sublist size is 1, then merges those sublists to produce a sorted list. The copy back step is avoided with alternating the direction of the merge with each level of recursion.



Merge sort animation. The elements to sort are represented by dots.

```

// Array A[] has the items to sort; array B[] is a work array.
TopDownMergeSort(A[], B[], n)
{
    CopyArray(A, 0, n, B);           // duplicate array A[] into B[]
    TopDownSplitMerge(B, 0, n, A);   // sort data from B[] into A[]
}

// Sort the given run of array A[] using array B[] as a source.
// iBegin is inclusive; iEnd is exclusive (A[iEnd] is not in the set).
TopDownSplitMerge(B[], iBegin, iEnd, A[])
{
    if(iEnd - iBegin < 2)             // if run size == 1
        return;                     // consider it sorted
    // split the run longer than 1 item into halves
    iMiddle = (iEnd + iBegin) / 2;    // iMiddle = mid point
    // recursively sort both runs from array A[] into B[]
    TopDownSplitMerge(A, iBegin, iMiddle, B); // sort the left run
    TopDownSplitMerge(A, iMiddle, iEnd, B);  // sort the right run
    // merge the resulting runs from array B[] into A[]
    TopDownMerge(B, iBegin, iMiddle, iEnd, A);
}

// Left source half is A[ iBegin:iMiddle-1].
// Right source half is A[iMiddle:iEnd-1 ].
// Result is B[ iBegin:iEnd-1 ].
TopDownMerge(A[], iBegin, iMiddle, iEnd, B[])
{
    i = iBegin, j = iMiddle;

    // While there are elements in the left or right runs...
    for (k = iBegin; k < iEnd; k++) {
        // If left run head exists and is <= existing right run head.
        if (i < iMiddle && (j >= iEnd || A[i] <= A[j])) {
            B[k] = A[i];
            i = i + 1;
        } else {
            B[k] = A[j];
            j = j + 1;
        }
    }
}

CopyArray(A[], iBegin, iEnd, B[])
{
    for(k = iBegin; k < iEnd; k++)
        B[k] = A[k];
}

```

## Bottom-up implementation

Example C-like code using indices for bottom up merge sort algorithm which treats the list as an array of  $n$  sublists (called *runs* in this example) of size 1, and iteratively merges sub-lists back and forth between two buffers:

```

// array A[] has the items to sort; array B[] is a work array
void BottomUpMergeSort(A[], B[], n)
{
    // Each 1-element run in A is already "sorted".
    // Make successively longer sorted runs of length 2, 4, 8, 16... until whole array is sorted.
    for (width = 1; width < n; width = 2 * width)
    {
        // Array A is full of runs of length width.
        for (i = 0; i < n; i = i + 2 * width)
        {
            // Merge two runs: A[i:i+width-1] and A[i+width:i+2*width-1] to B[]
            // or copy A[i:n-1] to B[] ( if(i+width >= n) )
            BottomUpMerge(A, i, min(i+width, n), min(i+2*width, n), B);
        }
        // Now work array B is full of runs of length 2*width.
        // Copy array B to array A for next iteration.
        // A more efficient implementation would swap the roles of A and B.
    }
}

```

```

        CopyArray(B, A, n);
        // Now array A is full of runs of length 2*width.
    }
}

// Left run is A[iLeft :iRight-1].
// Right run is A[iRight:iEnd-1 ].
BottomUpMerge(A[], iLeft, iRight, iEnd, B[])
{
    i = iLeft, j = iRight;
    // While there are elements in the left or right runs...
    for (k = iLeft; k < iEnd; k++) {
        // If left run head exists and is <= existing right run head.
        if (i < iRight && (j >= iEnd || A[i] <= A[j])) {
            B[k] = A[i];
            i = i + 1;
        } else {
            B[k] = A[j];
            j = j + 1;
        }
    }
}

void CopyArray(B[], A[], n)
{
    for(i = 0; i < n; i++)
        A[i] = B[i];
}

```

## Top-down implementation using lists

Pseudocode for top down merge sort algorithm which recursively divides the input list into smaller sublists until the sublists are trivially sorted, and then merges the sublists while returning up the call chain.

```

function merge_sort(list m)
    // Base case. A list of zero or one elements is sorted, by definition.
    if length of m ≤ 1 then
        return m

    // Recursive case. First, divide the list into equal-sized sublists
    // consisting of the first half and second half of the list.
    var left := empty list
    var right := empty list
    for each x with index i in m do
        if i ≤ (length of m)/2 then
            add x to left
        else
            add x to right

    // Recursively sort both sublists.
    left := merge_sort(left)
    right := merge_sort(right)

    // Then merge the now-sorted sublists.
    return merge(left, right)

```

In this example, the merge function merges the left and right sublists.

```

function merge(left, right)
    var result := empty list

    while left is not empty and right is not empty do
        if first(left) ≤ first(right) then
            append first(left) to result
            left := rest(left)
        else
            append first(right) to result
            right := rest(right)

```

```

// Either left or right may have elements left; consume them.
// (Only one of the following loops will actually be entered.)
while left is not empty do
    append first(left) to result
    left := rest(left)
while right is not empty do
    append first(right) to result
    right := rest(right)
return result

```

## Bottom-up implementation using lists

Pseudocode for bottom up merge sort algorithm which uses a small fixed size array of references to nodes, where  $\text{array}[i]$  is either a reference to a list of size  $2^i$  or 0. *node* is a reference or pointer to a node. The `merge()` function would be similar to the one shown in the top down merge lists example, it merges two already sorted lists, and handles empty lists. In this case, `merge()` would use *node* for its input parameters and return value.

```

function merge_sort(node head)
    // return if empty list
    if (head == nil)
        return nil
    var node array[32]; initially all nil
    var node result
    var node next
    var int i
    result = head
    // merge nodes into array
    while (result != nil)
        next = result.next;
        result.next = nil
        for(i = 0; (i < 32) && (array[i] != nil); i += 1)
            result = merge(array[i], result)
            array[i] = nil
        // do not go past end of array
        if (i == 32)
            i -= 1
        array[i] = result
        result = next
    // merge array into single list
    result = nil
    for (i = 0; i < 32; i += 1)
        result = merge(array[i], result)
    return result

```

## Natural merge sort

A natural merge sort is similar to a bottom up merge sort except that any naturally occurring runs (sorted sequences) in the input are exploited. Both monotonic and bitonic (alternating up/down) runs may be exploited, with lists (or equivalently tapes or files) being convenient data structures (used as FIFO queues or LIFO stacks).<sup>[3]</sup> In the bottom up merge sort, the starting point assumes each run is one item long. In practice, random input data will have many short runs that just happen to be sorted. In the typical case, the natural merge sort may not need as many passes because there are fewer runs to merge. In the best case, the input is already sorted (i.e., is one run), so the natural merge sort need only make one pass through the data. In many practical cases, long natural runs are present, and for that reason natural merge sort is exploited as the key component of Timsort. Example:

```

Start      : 3--4--2--1--7--5--8--9--0--6
Select runs : 3--4  2  1--7  5--8--9  0--6
Merge      : 2--3--4  1--5--7--8--9  0--6
Merge      : 1--2--3--4--5--7--8--9  0--6
Merge      : 0--1--2--3--4--5--6--7--8--9

```

Tournament replacement selection sorts are used to gather the initial runs for external sorting algorithms.

## Analysis

In sorting  $n$  objects, merge sort has an average and worst-case performance of  $O(n \log n)$ . If the running time of merge sort for a list of length  $n$  is  $T(n)$ , then the recurrence  $T(n) = 2T(n/2) + n$  follows from the definition of the algorithm (apply the algorithm to two lists of half the size of the original list, and add the  $n$  steps taken to merge the resulting two lists). The closed form follows from the master theorem.

In the worst case, the number of comparisons merge sort makes is equal to or slightly smaller than  $(n \lceil \lg n \rceil - 2^{\lceil \lg n \rceil} + 1)$ , which is between  $(n \lg n - n + 1)$  and  $(n \lg n + n + O(\lg n))$ .<sup>[4]</sup>

For large  $n$  and a randomly ordered input list, merge sort's expected (average) number of comparisons approaches  $\alpha \cdot n$  fewer than the worst case where

$$\alpha = -1 + \sum_{k=0}^{\infty} \frac{1}{2^k + 1} \approx 0.2645.$$

In the *worst* case, merge sort does about 39% fewer comparisons than quicksort does in the *average* case. In terms of moves, merge sort's worst case complexity is  $O(n \log n)$ —the same complexity as quicksort's best case, and merge sort's best case takes about half as many iterations as the worst case.

Merge sort is more efficient than quicksort for some types of lists if the data to be sorted can only be efficiently accessed sequentially, and is thus popular in languages such as Lisp, where sequentially accessed data structures are very common. Unlike some (efficient) implementations of quicksort, merge sort is a stable sort.

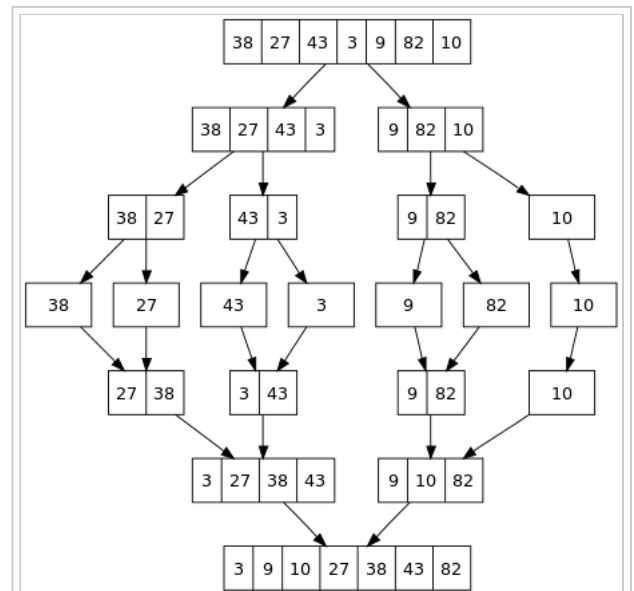
Merge sort's most common implementation does not sort in place;<sup>[5]</sup> therefore, the memory size of the input must be allocated for the sorted output to be stored in (see below for versions that need only  $n/2$  extra spaces).

## Variants

Variants of merge sort are primarily concerned with reducing the space complexity and the cost of copying.

A simple alternative for reducing the space overhead to  $n/2$  is to maintain *left* and *right* as a combined structure, copy only the *left* part of  $m$  into temporary space, and to direct the *merge* routine to place the merged output into  $m$ . With this version it is better to allocate the temporary space outside the *merge* routine, so that only one allocation is needed. The excessive copying mentioned previously is also mitigated, since the last pair of lines before the *return result* statement (function *merge* in the pseudo code above) become superfluous.

One drawback of merge sort, when implemented on arrays, is its  $O(n)$  working memory requirement. Several in-place variants have been suggested:



A recursive merge sort algorithm used to sort an array of 7 integer values. These are the steps a human would take to emulate merge sort (top-down).

- Katajainen *et al.* present an algorithm that requires a constant amount of working memory: enough storage space to hold one element of the input array, and additional space to hold  $O(1)$  pointers into the input array. They achieve an  $O(n \log n)$  time bound with small constants, but their algorithm is not stable.<sup>[6]</sup>
- Several attempts have been made at producing an *in-place merge* algorithm that can be combined with a standard (top-down or bottom-up) merge sort to produce an in-place merge sort. In this case, the notion of "in-place" can be relaxed to mean "taking logarithmic stack space", because standard merge sort requires that amount of space for its own stack usage. It was shown by Geffert *et al.* that in-place, stable merging is possible in  $O(n \log n)$  time using a constant amount of scratch space, but their algorithm is complicated and has high constant factors: merging arrays of length  $n$  and  $m$  can take  $5n + 12m + o(m)$  moves.<sup>[7]</sup> This high constant factor and complicated in-place algorithm was made simpler and easier to understand. Bing-Chao Huang and Michael A. Langston<sup>[8]</sup> presented a straightforward linear time algorithm *practical in-place merge* to merge a sorted list using fixed amount of additional space. They both have used the work of Kronrod and others. It merges in linear time and constant extra space. The algorithm takes little more average time than standard merge sort algorithms, free to exploit  $O(n)$  temporary extra memory cells, by less than a factor of two. Though the algorithm is much faster in practical way but it is unstable also for some list. But using similar concept they have been able to solve this problem. Other in-place algorithms include SymMerge, which takes  $O((n + m) \log (n + m))$  time in total.<sup>[9]</sup> Plugging such an algorithm into merge sort increases its complexity to the non-linearithmic, but still quasilinear,  $O(n \log^2 n)$ .

An alternative to reduce the copying into multiple lists is to associate a new field of information with each key (the elements in  $m$  are called keys). This field will be used to link the keys and any associated information together in a sorted list (a key and its related information is called a record). Then the merging of the sorted lists proceeds by changing the link values; no records need to be moved at all. A field which contains only a link will generally be smaller than an entire record so less space will also be used. This is a standard sorting technique, not restricted to merge sort.

## Use with tape drives

An external merge sort is practical to run using disk or tape drives when the data to be sorted is too large to fit into memory. External sorting explains how merge sort is implemented with disk drives. A typical tape drive sort uses four tape drives. All I/O is sequential (except for rewinds at the end of each pass). A minimal implementation can get by with just 2 record buffers and a few program variables.

Naming the four tape drives as A, B, C, D, with the original data on A, and using only 2 record buffers, the algorithm is similar to Bottom-up implementation, using pairs of tape drives instead of arrays in memory. The basic algorithm can be described as follows:

1. Merge pairs of records from A; writing two-record sublists alternately to C and D.
2. Merge two-record sublists from C and D into four-record sublists; writing these alternately to A and B.
3. Merge four-record sublists from A and B into eight-record sublists; writing these alternately to C and D.
4. Repeat until you have one list containing all the data, sorted --- in  $\log_2(n)$  passes.



Merge sort type algorithms allowed large data sets to be sorted on early computers that had small random access memories by modern standards. Records were stored on magnetic tape and processed on banks of magnetic tape drives, such as these IBM 729s.

Instead of starting with very short runs, usually a hybrid algorithm is used, where the initial pass will read many records into memory, do an internal sort to create a long run, and then distribute those long runs onto the output set. The step avoids many early passes. For example, an internal sort of 1024 records will save 9 passes. The internal sort is often large because it has such a benefit. In fact, there are techniques that can make the initial runs longer than the available internal memory.<sup>[10]</sup>

A more sophisticated merge sort that optimizes tape (and disk) drive usage is the polyphase merge sort.

## Optimizing merge sort

On modern computers, locality of reference can be of paramount importance in software optimization, because multilevel memory hierarchies are used. Cache-aware versions of the merge sort algorithm, whose operations have been specifically chosen to minimize the movement of pages in and out of a machine's memory cache, have been proposed. For example, the **tiled merge sort** algorithm stops partitioning subarrays when subarrays of size  $S$  are reached, where  $S$  is the number of data items fitting into a CPU's cache. Each of these subarrays is sorted with an in-place sorting algorithm such as insertion sort, to discourage memory swaps, and normal merge sort is then completed in the standard recursive fashion. This algorithm has demonstrated better performance on machines that benefit from cache optimization. (LaMarca & Ladner 1997)

Kronrod (1969) suggested an alternative version of merge sort that uses constant additional space. This algorithm was later refined. (Katajainen, Pasanen & Teuhola 1996)

Also, many applications of external sorting use a form of merge sorting where the input get split up to a higher number of sublists, ideally to a number for which merging them still makes the currently processed set of pages fit into main memory.

## Parallel merge sort

Merge sort parallelizes well due to use of the divide-and-conquer method. Several parallel variants are discussed in the third edition of Cormen, Leiserson, Rivest, and Stein's *Introduction to Algorithms*.<sup>[11]</sup> The first of these can be very easily expressed in a pseudocode with fork and join keywords:

```
// Sort elements lo through hi (exclusive) of array A.
algorithm mergesort(A, lo, hi) is
    if lo+1 < hi then // Two or more elements.
        mid =  $\lfloor (lo + hi) / 2 \rfloor$ 
        fork mergesort(A, lo, mid)
        mergesort(A, mid, hi)
    join
    merge(A, lo, mid, hi)
```

This algorithm is a trivial modification from the serial version, but its speedup is not impressive: when executed on an infinite number of processors, it runs in  $\Theta(n)$  time, which is only a  $\Theta(\log n)$  improvement on the serial version. A better result can be obtained by using a parallelized merge algorithm, which gives parallelism  $\Theta(n / \log^2 n)$ , meaning that this type of parallel merge sort runs in

$$\Theta\left(\frac{n \log n}{n \log^{-2} n}\right) = \Theta(\log^3 n)$$

time if enough processors are available.<sup>[11]</sup> Such a sort can perform well in practice when combined with a fast stable sequential sort, such as insertion sort, and a fast sequential merge as a base case for merging small arrays.<sup>[12]</sup>



Merge sort was one of the first sorting algorithms where optimal speed up was achieved, with Richard Cole using a clever subsampling algorithm to ensure  $O(1)$  merge.<sup>[13]</sup> Other sophisticated parallel sorting algorithms can achieve the same or better time bounds with a lower constant. For example, in 1991 David Powers described a parallelized quicksort (and a related radix sort) that can operate in  $O(\log n)$  time on a CRCW parallel random-access machine (PRAM) with  $n$  processors by performing partitioning implicitly.<sup>[14]</sup> Powers<sup>[15]</sup> further shows that a pipelined version of Batcher's Bitonic Mergesort at  $O(\log^2 n)$  time on a butterfly sorting network is in practice actually faster than his  $O(\log n)$  sorts on a PRAM, and he provides detailed discussion of the hidden overheads in comparison, radix and parallel sorting.

## Comparison with other sort algorithms

Although heapsort has the same time bounds as merge sort, it requires only  $\Theta(1)$  auxiliary space instead of merge sort's  $\Theta(n)$ . On typical modern architectures, efficient quicksort implementations generally outperform mergesort for sorting RAM-based arrays. On the other hand, merge sort is a stable sort and is more efficient at handling slow-to-access sequential media. Merge sort is often the best choice for sorting a linked list: in this situation it is relatively easy to implement a merge sort in such a way that it requires only  $\Theta(1)$  extra space, and the slow random-access performance of a linked list makes some other algorithms (such as quicksort) perform poorly, and others (such as heapsort) completely impossible.

As of Perl 5.8, merge sort is its default sorting algorithm (it was quicksort in previous versions of Perl). In Java, the `Arrays.sort()` (<http://docs.oracle.com/javase/6/docs/api/java/util/Arrays.html>) methods use merge sort or a tuned quicksort depending on the datatypes and for implementation efficiency switch to insertion sort when fewer than seven array elements are being sorted.<sup>[16]</sup> The linux kernel uses merge sort for its linked lists.<sup>[17]</sup> Python uses Timsort, another tuned hybrid of merge sort and insertion sort, that has become the standard sort algorithm in Java SE 7,<sup>[18]</sup> on the Android platform,<sup>[19]</sup> and in GNU Octave.<sup>[20]</sup>

## Notes

1. Knuth (1998, p. 158)
2. Jyrki Katajainen and Jesper Larsson Träff (1997). "A meticulous analysis of mergesort programs".
3. Powers, David M. W. and McMahon Graham B. (1983), "A compendium of interesting prolog programs", DCS Technical Report 8313, Department of Computer Science, University of New South Wales.
4. The worst case number given here does not agree with that given in Knuth's *Art of Computer Programming*, Vol 3. The discrepancy is due to Knuth analyzing a variant implementation of merge sort that is slightly sub-optimal
5. Cormen; Leiserson; Rivest; Stein. *Introduction to Algorithms*. p. 151. ISBN 978-0-262-03384-8.
6. Katajainen, Jyrki; Pasanen, Tomi; Teuhola, Jukka (1996). "Practical in-place mergesort". *Nordic J. Computing*. **3** (1): 27–40. CiteSeerX 10.1.1.22.8523 .
7. Geffert, Viliam; Katajainen, Jyrki; Pasanen, Tomi (2000). "Asymptotically efficient in-place merging". *Theoretical Computer Science*. **237**: 159–181. doi:10.1016/S0304-3975(98)00162-5.
8. Huang, Bing-Chao; Langston, Michael A. (March 1988). "Practical In-Place Merging". *Communications of the ACM*. **31** (3): 348–352. doi:10.1145/42392.42403.
9. Kim, Pok-Son; Kutzner, Arne (2004). *Stable Minimum Storage Merging by Symmetric Comparisons*. European Symp. Algorithms. Lecture Notes in Computer Science. **3221**. pp. 714–723. CiteSeerX 10.1.1.102.4612 . doi:10.1007/978-3-540-30140-0\_63. ISBN 978-3-540-23025-0.
10. Selection sort. Knuth's snowplow. Natural merge.
11. Cormen et al. 2009, pp. 797–805



12. V. J. Duvanenko, "Parallel Merge Sort", Dr. Dobbs's Journal, March 2011 (<http://drdobbs.com/high-performance-computing/229400239>)
13. Cole, Richard (August 1988). "Parallel merge sort". *SIAM J. Comput.* **17** (4): 770–785. doi:10.1137/0217049
14. Powers, David M. W. Parallelized Quicksort and Radixsort with Optimal Speedup (<http://citeseer.ist.psu.edu/327487.html>), *Proceedings of International Conference on Parallel Computing Technologies*. Novosibirsk. 1991.
15. David M. W. Powers, Parallel Unification: Practical Complexity (<http://david.wardpowers.info/Research/AI/papers/1995-01-ACAW-PUPC.pdf>), Australasian Computer Architecture Workshop, Flinders University, January 1995
16. OpenJDK Subversion (<https://openjdk.dev.java.net/source/browse/openjdk/jdk/trunk/jdk/src/share/classes/java/util/Arrays.java?view=markup>)
17. linux kernel /lib/list\_sort.c ([https://github.com/torvalds/linux/blob/master/lib/list\\_sort.c](https://github.com/torvalds/linux/blob/master/lib/list_sort.c))
18. jjb. "Commit 6804124: Replace "modified mergesort" in java.util.Arrays.sort with timsort". *Java Development Kit 7 Hg repo*. Retrieved 24 Feb 2011.
19. "Class: java.util.TimSort<T>". *Android JDK Documentation*. Archived from the original on January 20, 2015. Retrieved 19 Jan 2015.
20. "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."

## References

- Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2009) [1990]. *Introduction to Algorithms* (3rd ed.). MIT Press and McGraw-Hill. ISBN 0-262-03384-4.
- Katajainen, Jyrki; Pasanen, Tomi; Teuhola, Jukka (1996). "Practical in-place mergesort". *Nordic Journal of Computing*. **3**. pp. 27–40. ISSN 1236-6064. Retrieved 2009-04-04.. Also Practical In-Place Mergesort (<http://citeseer.ist.psu.edu/katajainen96practical.html>). Also [1] (<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.22.8523>)
- Knuth, Donald (1998). "Section 5.2.4: Sorting by Merging". *Sorting and Searching*. The Art of Computer Programming. **3** (2nd ed.). Addison-Wesley. pp. 158–168. ISBN 0-201-89685-0.
- Kronrod, M. A. (1969). "Optimal ordering algorithm without operational field". *Soviet Mathematics - Doklady*. **10**. p. 744.
- LaMarca, A.; Ladner, R. E. (1997). "The influence of caches on the performance of sorting". *Proc. 8th Ann. ACM-SIAM Symp. on Discrete Algorithms (SODA97)*: 370–379.
- Sun Microsystems. "Arrays API". Retrieved 2007-11-19.
- Sun Microsystems. "java.util.Arrays.java". Retrieved 2007-11-19.

## External links

- Animated Sorting Algorithms: Merge Sort (<http://www.ee.ryerson.ca/~courses/coe428/sorting/mergesort.html>) - Allows stepping through the steps in the algorithm.
- Dictionary of Algorithms and Data Structures: Merge sort (<https://xlinux.nist.gov/dads/HTML/mergesort.html>)
- Mergesort applet (<http://www.yorku.ca/sychen/research/sorting/index.html>) with "level-order" recursive calls to help improve algorithm analysis
- Open Data Structures - Section 11.1.1 - Merge Sort ([http://opendatastructures.org/versions/edition-0.1e/ods-java/11\\_1\\_Comparison\\_Based\\_Sort.html#SECTION00141100000000000000](http://opendatastructures.org/versions/edition-0.1e/ods-java/11_1_Comparison_Based_Sort.html#SECTION00141100000000000000))



The Wikibook *Algorithm implementation* has a page on the topic of: **Merge sort**

Retrieved from "[https://en.wikipedia.org/w/index.php?title=Merge\\_sort&oldid=747460129](https://en.wikipedia.org/w/index.php?title=Merge_sort&oldid=747460129)"

Categories: Sorting algorithms | Comparison sorts | Stable sorts

- This page was last modified on 2 November 2016, at 14:23.

- Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy. Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.