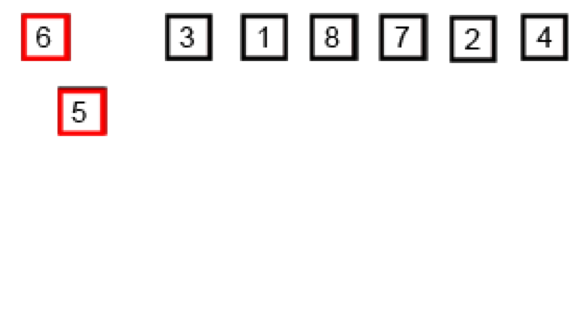
Merge sort

From Wikipedia, the free encyclopedia

In computer science, **merge sort** (also commonly spelled **mergesort**) is an $O(n \log n)$ 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.[1] A detailed description and analysis of bottom-up mergesort appeared in a report by Goldstine and Neumann as early as 1948.[2]

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.

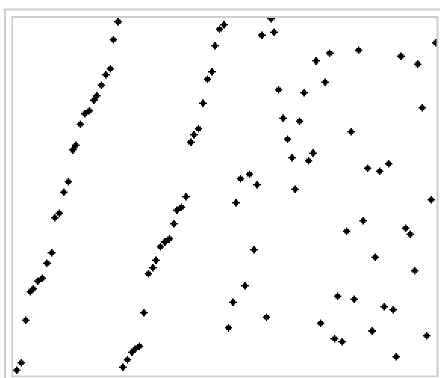
Class	Sorting algorithm
Data structure	Array
Worst case performance	O(n log n)
Best case performance	O(n log n) typical, O(n) natural variant
Average case performance	O(n log n)
Worst case space complexity	O(n) auxiliary

Contents

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.



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

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 could be avoided if the recursion alternated between two functions so that the direction of the merge corresponds with the level of recursion.

```
TopDownMergeSort(A[], B[], n)
{
    TopDownSplitMerge(A, 0, n, B);
}

// iBegin is inclusive; iEnd is exclusive (A[iEnd] is not in the set)
TopDownSplitMerge(A[], iBegin, iEnd, B[])
{
    if(iEnd - iBegin < 2)                       // if run size == 1
        return;                                 //   consider it sorted
    // recursively split runs into two halves until run size == 1,
    // then merge them and return back up the call chain
    iMiddle = (iEnd + iBegin) / 2;              // iMiddle = mid point
    TopDownSplitMerge(A, iBegin,  iMiddle, B);  // split / merge left  half
    TopDownSplitMerge(A, iMiddle,    iEnd, B);  // split / merge right half
    TopDownMerge(A, iBegin, iMiddle, iEnd, B);  // merge the two half runs
    CopyArray(B, iBegin, iEnd, A);              // copy the merged runs back to A
}

//  left half is A[iBegin :iMiddle-1]
// right half is A[iMiddle:iEnd-1   ]
TopDownMerge(A[], iBegin, iMiddle, iEnd, B[])
{
    i0 = iBegin, i1 = iMiddle;

    // While there are elements in the left or right runs
    for (j = iBegin; j < iEnd; j++) {
        // If left run head exists and is <= existing right run head.
        if (i0 < iMiddle && (i1 >= iEnd || A[i0] <= A[i1]))
            B[j] = A[i0];
            i0 = i0 + 1;
        else
            B[j] = A[i1];
            i1 = i1 + 1;
    }
}

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

```
void
BottomUpMerge(A[], iLeft, iRight, iEnd, B[])
{
  i0 = iLeft;
  i1 = iRight;
  j;

  /* While there are elements in the left or right runs */
  for (j = iLeft; j < iEnd; j++)
    {
      /* If left run head exists and is <= existing right run head */
      if (i0 < iRight && (i1 >= iEnd || A[i0] <= A[i1]))
        {
          B[j] = A[i0];
          i0 = i0 + 1;
        }
      else
        {
          B[j] = A[i1];
          i1 = i1 + 1;
        }
    }
```

```
}

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

/* array A[] has the items to sort; array B[] is a work array */
void
BottomUpSort(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. */
    }
}
```

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(m) <= 1
        return m

    // Recursive case. First, *divide* the list into equal-sized sublists.
    var list left, right
    var integer middle = length(m) / 2
    for each x in m before middle
         add x to left
    for each x in m after or equal middle
         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 list result
    while notempty(left) and notempty(right)
        if first(left) <= first(right)
            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
    while notempty(left)
        append first(left) to result
        left = rest(left)
    while notempty(right)
        append first(right) to result
        right = rest(right)
    return result
```

Natural merge sort

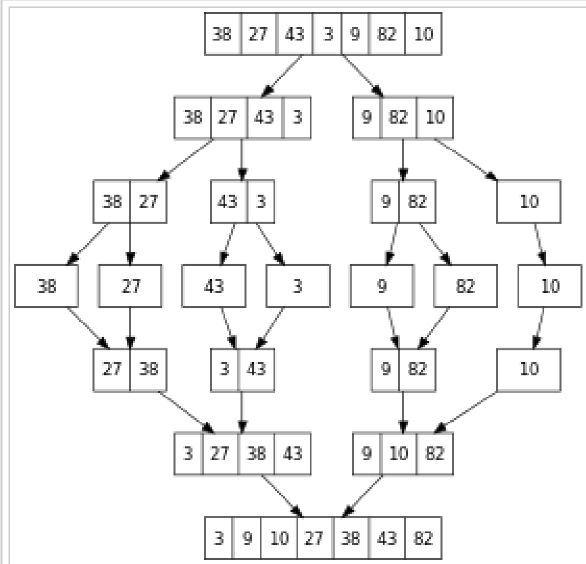
A natural merge sort is similar to a bottom up merge sort except that any naturally occurring runs (sorted sequences) in the input are exploited. In the bottom up merge sort, the starting point assumes each run is one item long. In practice, random input data will have many short runs that just happen 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. 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.



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).

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 $(n \lg n + n + O(\lg n))$.[3]

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; 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).

Merge sort also has some demerits. One is its use of $2n$ locations; the additional n locations are commonly used because merging two sorted sets in place is more complicated and would need more comparisons and move operations. But despite the use of this space the algorithm still does a lot of work: The contents of *m* are first copied into *left* and *right* and later into the list *result* on each invocation of *merge_sort* (variable names according to the pseudocode above).

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.

In-place sorting is possible, and still stable, but is more complicated, and slightly slower, requiring non-linearithmic quasilinear time $O(n \log^2 n)$ One way to sort in-place is to merge the blocks recursively.[4] Like the standard merge sort, in-place merge sort is also a stable sort. In-place stable sorting of linked lists is simpler. In this case the algorithm does not use more space than that already used by the list representation, but the $O(\log(k))$ used for the recursion trace.

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.

A variant named *binary merge sort* uses a *binary insertion sort* to sort groups of 32 elements, followed by a final sort using merge sort. It combines the speed of insertion sort on small data sets with the speed of merge sort on large data sets.[5]

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.

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.

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 log2(n) passes.

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.[6]

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*.[7] The first of these can be very easily expressed in a pseudocode with fork and join keywords:

```
/* inclusive/exclusive indices */
procedure mergesort(A, lo, hi):
    if lo+1 < hi:  /* if two or more elements */
        mid = ⌊(lo + hi) / 2⌋
        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: $\Theta(\log n)$. A parallel merge algorithm to not only parallelize the recursive division of the array, but also the merge operation leads to a better parallel sort that performs 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.[8] 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.[9] 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 PRAM with n processors by performing partitioning implicitly.[10] Powers[11] 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 Θ(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.[12] Python uses Timsort, another tuned hybrid of merge sort and insertion sort, that has become the standard sort algorithm in Java SE 7,[13] on the Android platform,[14] and in GNU Octave.[15]

Notes

1. ^ Knuth (1998, p. 158)
2. ^ Jyrki Katajainen and Jesper Larsson Träff (1997). "A meticulous analysis of mergesort programs".
3. ^ 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
4. ^ A Java implementation of in-place stable merge sort (http://h2database.googlecode.com/svn/trunk/h2/src/tools/org/h2/dev/sort/InPlaceStableMergeSort.java)
5. ^ "Binary Merge Sort" (https://docs.google.com/file/d/0B8KIVX-AaaGiYzcta0pFUXJnNG8).
6. ^ Selection sort. Knuth's snowplow. Natural merge.
7. ^ Cormen et al. 2009, pp. 797–805
8. ^ V. J. Duvanenko, "Parallel Merge Sort", Dr. Dobb's Journal, March 2011 (http://drdobbs.com/high-performance-computing/229400239)
9. ^ Cole, Richard (August 1988). "Parallel merge sort". *SIAM J. Comput*. **17** (4): 770–785. doi:10.1137/0217049 (https://dx.doi.org/10.1137%2F0217049)
10. ^ 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.
11. ^ David M. W. Powers, Parallel Unification: Practical Complexity (http://david.wardpowers.info/Research/AI/papers/199501-ACAW-PUPC.pdf), Australasian Computer Architecture Workshop, Flinders University, January 1995
12. ^ OpenJDK Subversion (https://openjdk.dev.java.net/source/browse/openjdk/jdk/trunk/jdk/src/share/classes/java/util/Arrays.java?view=markup)
13. ^ jjb. "Commit 6804124: Replace "modified mergesort" in java.util.Arrays.sort with timsort" (http://hg.openjdk.java.net/jdk7/tl/jdk/rev/bfd7abda8f79). *Java Development Kit 7 Hg repo*. Retrieved 24 Feb 2011.
14. ^ "Class: java.util.TimSort<T>" (https://android.googlesource.com/platform/libcore/+/jb-mr2-release/luni/src/main/java/java/util/TimSort.java). *Android JDK Documentation*. Retrieved 19 Jan 2015.
15. ^ "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 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" (http://www.diku.dk/hjemmesider/ansatte/jyrki/Paper/mergesort_NJC.ps). *Nordic Journal of Computing* **3**. pp. 27–

40. ISSN 1236-6064 (https://www.worldcat.org/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" (http://java.sun.com/javase/6/docs/api/java/util/Arrays.html). Retrieved 2007-11-19.
- Sun Microsystems. "java.util.Arrays.java" (https://openjdk.dev.java.net/source/browse/openjdk/jdk/trunk/jdk/src/share/classes/java/util/Arrays.java?view=markup). Retrieved 2007-11-19.

External links

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

- Animated Sorting Algorithms: Merge Sort (http://www.sorting-algorithms.com/merge-sort) – graphical demonstration and discussion of array-based merge sort
- Dictionary of Algorithms and Data Structures: Merge sort (http://www.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_Sorti.html#SECTION001411000000000000000)

Retrieved from "http://en.wikipedia.org/w/index.php?title=Merge_sort&oldid=648626827"

Categories: Sorting algorithms | Comparison sorts | Stable sorts

- This page was last modified on 24 February 2015, at 13:42.