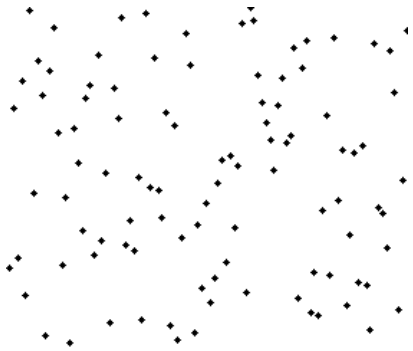
Merge sort



Example of merge sort sorting a list of random dots.

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

Merge sort is an *O*(*n* log *n*) comparison-based sorting algorithm. Most implementations produce a stable sort, meaning that the implementation preserves the input order of equal elements in the sorted output. It is a divide and conquer algorithm. Merge sort was invented by John von Neumann in 1945.[1]

Algorithm

Conceptually, a merge sort works as follows

1. If the list is of length 0 or 1, then it is already sorted. Otherwise:
2. Divide the unsorted list into two sublists of about half the size.
3. Sort each sublist recursively by re-applying merge sort.
4. Merge the two sublists back into one sorted list.

Merge sort incorporates two main ideas to improve its runtime:

1. A small list will take fewer steps to sort than a large list.
2. Fewer steps are required to construct a sorted list from two sorted lists than two unsorted lists. For example, you only have to traverse each list once if they're already sorted (see the merge function below for an example implementation).

Example: Using merge sort to sort a list of integers contained in an array:

Suppose we have an array *A* with *n* indices ranging from A_0 to A_{n-1}. We apply merge sort to $A(A_0..A_{c-1})$ and $A(A_c..A_{n-1})$ where *c* is the integer part of $n/2$. When the two halves are returned they will have been sorted. They can now be merged together to form a sorted array.
In a simple pseudocode form, the algorithm could look something like this:

```
function merge_sort(m)
    if length(m) ≤ 1
        return m
    var list left, right, result
```

```
    var integer middle = length(m) / 2
    for each x in m up to middle
         add x to left
    for each x in m after middle
         add x to right
    left = merge_sort(left)
    right = merge_sort(right)
    result = merge(left, right)
    return result
```

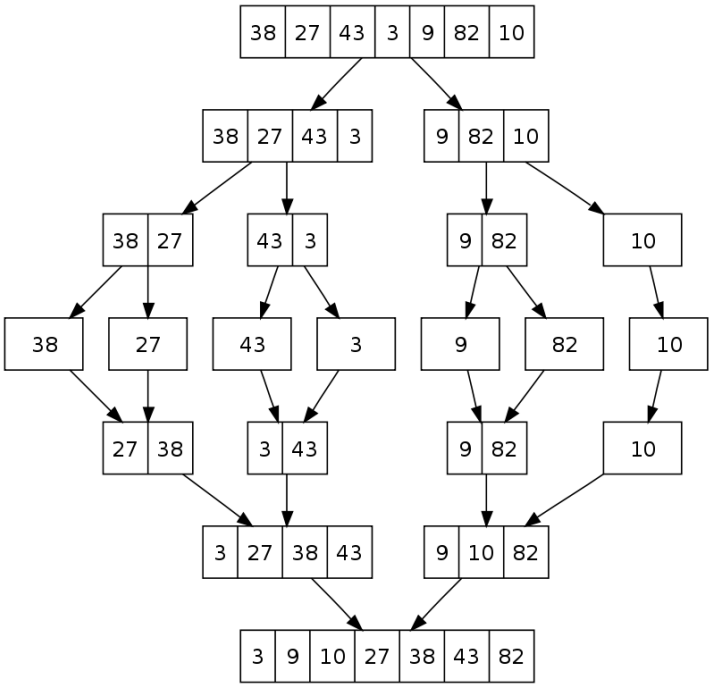
Following writing merge_sort function, then it is required to merge both the left and right lists created above. There are several variants for the merge() function; one possibility is this:

```
function merge(left,right)
    var list result
    while length(left) > 0 or length(right) > 0
        if length(left) > 0 and length(right) > 0
            if first(left) ≤ first(right)
                append first(left) to result
                left = rest(left)
            else
                append first(right) to result
                right = rest(right)
        else if length(left) > 0
            append first(left) to result
            left = rest(left)
        else if length(right) > 0
            append first(right) to result
            right = rest(right)
    end while
    return result
```

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*) = 2*T*(*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, merge sort does an amount of comparisons 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 ($n \lg n + n + O(\lg n)$).[2]



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).

For large *n* and a randomly ordered input list, merge sort's expected (average) number of comparisons approaches *α*·*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; merge sort always makes fewer comparisons than quicksort, except in extremely rare cases, when they tie, where merge sort's *worst* case is found simultaneously with quicksort's *best* 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.

Recursive implementations of merge sort make 2*n* − 1 method calls in the worst case, compared to quicksort's *n*, thus merge sort has roughly twice as much recursive overhead as quicksort. However, iterative, non-recursive implementations of merge sort, avoiding method call overhead, are not difficult to code. 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 as described here also has an often overlooked, but practically important, best-case property. If the input is already sorted, its complexity falls to O(*n*). Specifically, *n-1* comparisons and zero moves are performed, which is the same as for simply running through the input, checking if it is pre-sorted.

Sorting in-place is possible (e.g., using lists rather than arrays) but is very complicated, and will offer little performance gains in practice, even if the algorithm runs in O(*n* log *n*) time. (Katajainen, Pasanen & Teuhola 1996) In these cases, algorithms like heapsort usually offer comparable speed, and are far less complex. Additionally, unlike the standard merge sort, in-place merge sort is not a stable sort. In the case of linked lists the algorithm does not use more space than that the already used by the list representation, but the O(log(k)) used for the recursion trace.

Merge sort is more efficient than quick sort 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 as long as the merge operation is implemented properly.

As can be seen from the procedure merge sort, there are some demerits. One complaint we might raise is its use of 2*n* locations; the additional *n* locations were needed because one couldn't reasonably merge two sorted sets in place. But despite the use of this space the algorithm must still work hard: 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). An alternative to this copying 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.

Another 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 in the previous paragraph is also mitigated, since the last pair of lines before the *return result* statement (function *merge* in the pseudo code above) become superfluous.

Merge sort using tape drives

An external merge sort is practical to run using tape drives as input and output devices. It requires very little memory, and the memory required does not depend on the number of records.

For the same reason it is also useful for sorting data on disk that is too large to fit entirely into primary memory. On tape drives that can run both backwards and forwards, merge passes can be run in both directions, avoiding rewind time.

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.

If you have four tape drives, it works as follows:

1. Divide the data to be sorted in half and put half on each of two tapes
2. Merge individual pairs of records from the two tapes; write two-record chunks alternately to each of the two output tapes
3. Merge the two-record chunks from the two output tapes into four-record chunks; write these alternately to the original two input tapes
4. Merge the four-record chunks into eight-record chunks; write these alternately to the original two output tapes
5. Repeat until you have one chunk containing all the data, sorted --- that is, for log *n* passes, where *n* is the number of records.

For almost-sorted data on tape, a bottom-up "natural merge sort" variant of this algorithm is popular.

The bottom-up "natural merge sort" merges whatever "runs" of in-order records are already in the data. In the worst case (reversed data), "natural merge sort" performs the same as the above—it merges individual records into 2-record chunks, then 2-record chunks into 4-record chunks, etc. In the best case (already mostly-sorted data), "natural merge sort" merges large already-sorted chunks into even larger chunks, hopefully finishing in fewer than log *n* passes.

In a simple pseudocode form, the "natural merge sort" algorithm could look something like this:

```
# Original data is on the input tape; the other tapes are blank
function merge_sort(input_tape, output_tape, scratch_tape_C, scratch_tape_D)
    while any records remain on the input_tape
        while any records remain on the input_tape
            merge( input_tape, output_tape, scratch_tape_C)
            merge( input_tape, output_tape, scratch_tape_D)
        while any records remain on C or D
            merge( scratch_tape_C, scratch_tape_D, output_tape)
            merge( scratch_tape_C, scratch_tape_D, input_tape)
```

```
# take the next sorted run from the input tapes, and merge into the single given output_tape.
# tapes are scanned linearly.
# tape[next] gives the record currently under the read head of that tape.
# tape[current] gives the record previously under the read head of that tape.
# (Generally both tape[current] and tape[previous] are buffered in RAM ...)
function merge(left[], right[], output_tape[])
    do
       if left[current] ≤ right[current]
           append left[current] to output_tape
           read next record from left tape
       else
           append right[current] to output_tape
           read next record from right tape
   while left[current] < left[next] and right[current] < right[next]
   if left[current] < left[next]
       append current_left_record to output_tape
   if right[current] < right[next]
       append current_right_record to output_tape
   return
```

Either form of merge sort can be generalized to any number of tapes.

A more sophisticated merge sort is the polyphase merge sort.

Optimizing merge sort

On modern computers, locality of reference can be of paramount importance in software optimization, because multi-level 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 single page in memory. Each of these subarrays is sorted with an in-place sorting algorithm, 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 refined by Katajainen, Pasanen & Teuhola (1996).

Comparison with other sort algorithms

Although heapsort has the same time bounds as merge sort, it requires only Θ(1) auxiliary space instead of merge sort's Θ(*n*), and is often faster in practical implementations. 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, parallelizes better, 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() [3] 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.[4] Python uses timsort, another tuned hybrid of merge sort and insertion sort, which will also become the standard sort algorithm for Java SE 7.[5]

Utility in online sorting

Merge sort's merge operation is useful in online sorting, where the list to be sorted is received a piece at a time, instead of all at the beginning. In this application, we sort each new piece that is received using any sorting algorithm, and then merge it into our sorted list so far using the merge operation. However, this approach can be expensive in time and space if the received pieces are small compared to the sorted list — a better approach in this case is to store the list in a self-balancing binary search tree and add elements to it as they are received.

Notes

[1] Merge Sort - Wolfram MathWorld (http://mathworld.wolfram.com/MergeSort.html)

[2] 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

[3] http://java.sun.com/j2se/latest/docs/api/java/util/Arrays.html

[4] OpenJDK Subversion (https://openjdk.dev.java.net/source/browse/openjdk/jdk/trunk/jdk/src/share/classes/java/util/Arrays.java?view=markup)

[5] http://hg.openjdk.java.net/jdk7/tl/jdk/rev/bfd7abda8f79

References

- Cormen, Thomas H.; Leiserson, Charles E., Rivest, Ronald L., Stein, Clifford (2001) [1990]. "2.3: Designing algorithms". *Introduction to Algorithms* (2nd ed.). MIT Press and McGraw-Hill. pp. pp. 27–37. ISBN 0-262-03293-7.
- 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. Retrieved 2009-04-04. Also Practical In-Place Mergesort (http://citeseer.ist.psu.edu/katajainen96practical.html). Also (http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.22.8523)
- Knuth, Donald (1998). "Section 5.2.4: Sorting by Merging". *The Art of Computer Programming*. Addison-Wesley. pp. 158–168. ISBN 0-201-89685-0.
- Kronrod, M. A. (1969). "Optimal ordering algorithm without operational field". *Soviet Mathematics - Doklady* **10**: pp. 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, Inc.. "Arrays API" (http://java.sun.com/javase/6/docs/api/java/util/Arrays.html). Retrieved 2007-11-19.

- Sun Microsystems, Inc.. "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

- Merge sort in 20 languages (http://www.codecodex.com/wiki/Merge_sort)
- Animated Sorting Algorithms: Merge Sort (http://www.sorting-algorithms.com/merge-sort) – graphical demonstration and discussion of array-based merge sort
- Merge sort applet (http://www.atkinson.yorku.ca/~sychen/research/sorting/sortingHome.html) with level order recursive calls to help improve algorithm analysis
- Dictionary of Algorithms and Data Structures: Merge sort (http://www.nist.gov/dads/HTML/mergesort.html)
- Implementation of merge sort in various languages (http://www.rosettacode.org/wiki/Merge_sort) on Rosetta Code
- Literate implementations of merge sort in various languages (http://en.literateprograms.org/Category:Merge_sort) on LiteratePrograms
- A colored graphical Java applet (http://coderaptors.com/?MergeSort) which allows experimentation with initial state and shows statistics
- Simon Tatham's explanation and code for a merge sort (http://www.chiark.greenend.org.uk/~sgtatham/algorithms/listsort.html)
- MergeSort tutorial and Java code for beginners (http://www.mycstutorials.com/articles/sorting/mergesort)
- Merge sort Fortran routines (http://www.fortran-2000.com/rank/)

Article Sources and Contributors

Merge sort *Source*: http://en.wikipedia.org/w/index.php?oldid=405986390 *Contributors*: 192.58.206.xxx, 209.157.137.xxx, Ablonus, Abu adam, Adambro, Ahmadsh, Ahy1, Alain Amiouni, Alansohn, Allan McInnes, Amahdy, Amiodusz, Andrei Stroe, Andy M. Wang, Antientropic, Apoorbo, ArnoldReinhold, Artagnon, Avb, Bakanov, Base698, Bayard, Bill wang1234, Black Falcon, Bobrayner, Booyabazooka, BrokenSegue, Bubba73, C. A. Russell, CJLL Wright, CRGreathouse, Cactus.man, Caesura, Captain Fortran, Ceros, Cic, CobaltBlue, Comocomocomocomo, Conversion script, Cuzelac, Cybercobra, DFRussia, Dammit, Damonkohler, Danakil, Daniel Geisler, Daniel Quinlan, Daztekk, Dbagnall, Dcoetzee, Deanonwiki, Decrypt3, Delldot, Destynova, DevastatorIIC, Dima1, Discospinster, Donhalcon, Dr. Gonzo, Duckbill, EAspenwood, Easwarno1, Eleschinski2000, Erel Segal, Eric119, Eserra, Ewlyahoocom, Fashnek, Fizo86, Fred J, Fredrik, Frencheigh, Fyyer, Garas, Garo, Garrettw87, Garyzx, Giftlite, Glrx, GraemeL, GregorB, Haham hanuka, Hariva, Hoof1341, Hyad, Immunize, Intgr, Itmozart, J.delanoy, JF Bastien, Jacobolus, Jafet, Jengelh, Jirka6, Jleedev, Jogloran, JohnOwens, Josh Kehn, Joshk, Jpl, Kalebdf, Kbk, Kenb215, Kenyon, Klrste, Knutux, Kragen, Lee Daniel Crocker, Mattjohnson, Mav, Mecej4, Michele bon, Mikeblas, Mikewebkist, Minesweeper, Mipadi, Miskaton, MisterSheik, Mlhetland, Mntlchaos, Mutinus, N26ankur, Naku, Negrulio, Neilc, Nguyen Thanh Quang, Nickls, Ninjatummen, Nixeagle, Nmnogueira, NotARusski, Novas0x2a, Nuno Tavares, Nyenyec, Octotron, Ohnoitsjamie, Oli Filth, Onco p53, Onevalefan, OranL, Orderud, Oskar Sigvardsson, Oxaric, Oğuz Ergin, PavelY, Pfalstad, Pkrecker, Quasipalm, Radiozilla, Renku, ReyBrujo, Rfl, Rhanekom, Rich Farmbrough, Rl, Romanm, Rspeer, Ruud Koot, SLi, Sam nead, Sanjay742, SashaMarievskaya, Schorzman78, Scott Paeth, ScottBurson, Shadowjams, Shashwat2691, Shellreef, Silly rabbit, SiobhanHansa, Sir Nicholas de Mimsy-Porpington, Sirex98, Sligocki, Soultaco, Sperling, StuartBrady, Swift, Sychen, T0m, TakuyaMurata, The Anome, Thijswijs, Thue, Timwi, TobiasPersson, Tohd8BohaithuGh1, Tomchiukc, Tommunist, UncleDouggie, Vexis, VineetKumar, Vorn, Wilagobler, WojciechSwiderski, Worch, Ww, Xhackeranywhere, Xueshengyao, Zhaladshar, Zophar1, 368 anonymous edits

Image Sources, Licenses and Contributors

Image:Merge sort animation2.gif *Source*: http://en.wikipedia.org/w/index.php?title=File:Merge_sort_animation2.gif *License*: Creative Commons Attribution-Sharealike 2.5 *Contributors*: CobaltBlue

Image:merge sort algorithm diagram.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Merge_sort_algorithm_diagram.svg *License*: Public Domain *Contributors*: Original uploader was VineetKumar at en.wikipedia

File:IBM 729 Tape Drives.nasa.jpg *Source*: http://en.wikipedia.org/w/index.php?title=File:IBM_729_Tape_Drives.nasa.jpg *License*: Public Domain *Contributors*: NASA

License