**Merge sort**

**Merge sort** is a [sorting](http://www.algorithmist.com/index.php/Sorting) algorithm invented by John von Neumann based on the [divide and conquer](http://www.algorithmist.com/index.php/Divide_and_Conquer) technique. It always runs in \Theta(n \log n)\,time, but requires O(n)\,space. The general concept is that we first break the list into two smaller lists of roughly the same size, and then use merge sort recursively on the subproblems, until they cannot subdivide anymore (i.e. when they contain zero or one elements). Then, we can merge by stepping through the lists in linear time. The recurrence is thus:

T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + \Theta(n)

## Pseudo-code

*a* is an array containing *n* elements.

func mergesort( var a as array )

if ( n == 1 ) return a

var l1 as array = a[0] ... a[n/2]

var l2 as array = a[n/2+1] ... a[n]

l1 = mergesort( l1 )

l2 = mergesort( l2 )

return merge( l1, l2 )

end func

func merge( var a as array, var b as array )

var c as array

while ( a and b have elements )

if ( a[0] > b[0] )

add b[0] to the end of c

remove b[0] from b

else

add a[0] to the end of c

remove a[0] from a

while ( a has elements )

add a[0] to the end of c

remove a[0] from a

while ( b has elements )

add b[0] to the end of c

remove b[0] from b

return c

end func

## Bottom-up merge sort

**Bottom-up merge sort** is a non-recursive variant of the **merge sort**, in which the array is sorted by a sequence of passes. During each pass, the array is divided into blocks of size m\,. (Initially, m=1\,). Every two adjacent blocks are merged (as in normal merge sort), and the next pass is made with a twice larger value of m\,.

In pseudo-code:

Input: array a[] indexed from 0 to n-1.

m = 1

while m < n do

i = 0

while i < n-m do

merge subarrays a[i..i+m-1] and a[i+m .. min(i+2\*m-1,n-1)] in-place.

i = i + 2 \* m

m = m \* 2

## Natural mergesort

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

The bottom-up "natural mergesort" merges whatever "chunks" of in-order records are already in the data. In the worst case (reversed data), "natural mergesort" 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 mergesort" merges large already-sorted chunks into even larger chunks, hopefully finishing in fewer than log *n* passes.

Pseudocode

# Original data is on the input tape; the other tapes are blank

**function** mergesort(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 chunk 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**