http://www.personal.kent.edu/%7Ermuhamma/Maingif/redline.gif

**M**erge sort is based on the **divide-and-conquer** paradigm. Its worst-case running time has a lower order of growth than insertion sort. Since we are dealing with subproblems, we state each subproblem as sorting a subarray A[p .. r]. Initially, p = 1 and r = n, but these values change as we recurse through subproblems.

To sort A[p .. r]:

1. **Divide Step**

If a given array A has zero or one element, simply return; it is already sorted. Otherwise, split A[p .. r] into two subarrays A[p .. q] and A[q + 1 .. r], each containing about half of the elements of A[p .. r]. That is, q is the halfway point of A[p .. r].

2. **Conquer Step**

Conquer by recursively sorting the two subarrays A[p .. q] and A[q + 1 .. r].

3. **Combine Step**

Combine the elements back in A[p .. r] by merging the two sorted subarrays A[p .. q] and A[q + 1 .. r] into a sorted sequence. To accomplish this step, we will define a procedure MERGE (A, p, q, r).

Note that the recursion bottoms out when the subarray has just one element, so that it is trivially sorted.

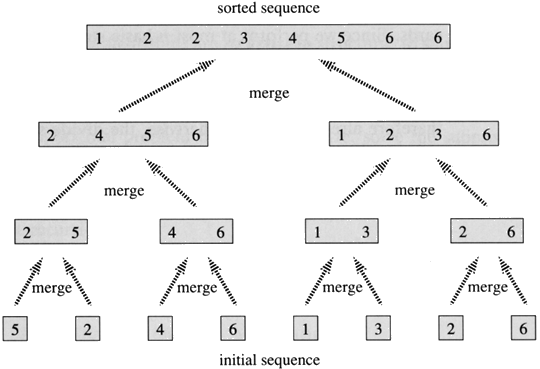
**Algorithm: Merge Sort**

To sort the entire sequence A[1 .. n], make the initial call  to the procedure MERGE-SORT (A, 1, n).

MERGE-SORT (A, p, *r*)

1.     IF p < r                                                    // Check for base case  
2.         THEN q = FLOOR[(p + r)/2]                 // Divide step  
3.                 MERGE (A, p, q)                          // Conquer step.  
4.                 MERGE (A, q + 1, r)                     // Conquer step.  
5.                 MERGE (A, p, q, r)                       // Conquer step.

Example: Bottom-up view of the above procedure for n = 8.



**Merging**

What remains is the MERGE procedure. The following is the input and output of the MERGE procedure.

**INPUT**: Array A and indices p, q, r such that p ≤ q ≤ r and subarray A[p .. q] is sorted and subarray A[q + 1 .. r] is sorted. By restrictions on p, q, r, neither subarray is empty.

**OUTPUT**: The two subarrays are merged into a single sorted subarray in A[p .. r].

We implement it so that it takes Θ(n) time, where n = r − p + 1, which is the number of elements being merged.

**Idea Behind Linear Time Merging**

Think of two piles of cards, Each pile is sorted and placed face-up on a table with the smallest cards on top. We will merge these into a single sorted pile, face-down on the table.

A basic step:

* Choose the smaller of the two top cards.
* Remove it from its pile, thereby exposing a new top card.
* Place the chosen card face-down onto the output pile.
* Repeatedly perform basic steps until one input pile is empty.
* Once one input pile empties, just take the remaining input pile and place it face-down onto the output pile.

Each basic step should take constant time, since we check just the two top cards. There are at most n basic steps, since each basic step removes one card from the input piles, and we started with n cards in the input piles. Therefore, this procedure should take Θ(n) time.

Now the question is do we actually need to check whether a pile is empty before each basic step?   
The answer is no, we do not. Put on the bottom of each input pile a special **sentinel** card. It contains a special value that we use to simplify the code. We use ∞, since that's guaranteed to lose to any other value. The only way that ∞ cannot lose is when both piles have ∞ exposed as their top cards. But when that happens, all the nonsentinel cards have already been placed into the output pile. We know in advance that there are exactly r − p + 1 nonsentinel cards so stop once we have performed r − p + 1 basic steps. Never a need to check for sentinels, since they will always lose. Rather than even counting basic steps, just fill up the output array from index p up through and including index r .

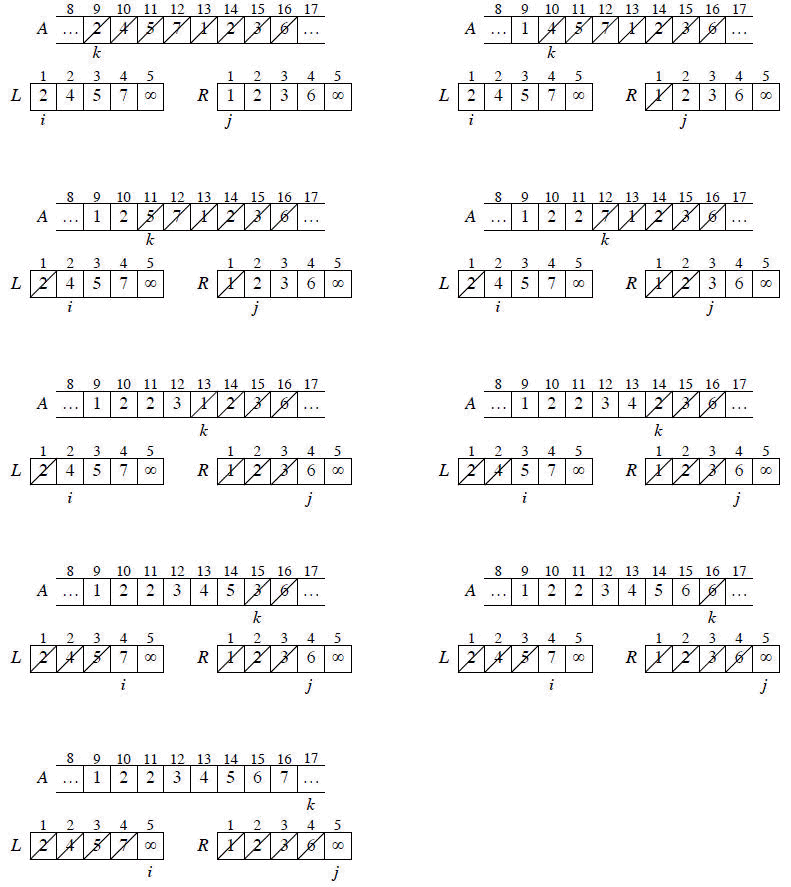
The **pseudocode** of the MERGE procedure is as follow:

MERGE (A, p, q, r )

1.      n1 ← q − p + 1  
2.      n2 ← r − q  
3.      Create arrays L[1 . . n1 + 1] and R[1 . . n2 + 1]  
4.      **FOR** i ← 1 **TO** n1  
5.            **DO** L[i] ← A[p + i − 1]  
6.      **FOR** j ← 1 **TO** n2  
7.            **DO** R[j] ← A[q + j ]  
8.      L[n1 + 1] ← ∞  
9.      R[n2 + 1] ← ∞  
10.    i ← 1  
11.    j ← 1  
12.    **FOR** k ← p **TO** r  
13.         **DO IF** L[i ] ≤ R[ j]  
14.                **THEN** A[k] ← L[i]  
15.                        i ← i + 1  
16.                **ELSE** A[k] ← R[j]  
17.                        j ← j + 1

**Example** [from CLRS-Figure 2.3]: A call of MERGE(A, 9, 12, 16). **Read the following figure row by row**. That is how we have done in the class.

* The first part shows the arrays at the start of the "for k ← p to r" loop, where A[p . . q] is copied into L[1 . . n1] and A[q + 1 . . r ] is  
  copied into R[1 . . n2].
* Succeeding parts show the situation at the start of successive iterations.
* Entries in A with slashes have had their values copied to either L or R and have not had a value copied back in yet. Entries in L and R with slashes have been copied back into A.
* The last part shows that the subarrays are merged back into A[p . . r], which is now sorted, and that only the sentinels (∞) are exposed in the arrays L and R.]



**Running Time**

The first two **for** loops (that is, the loop in line 4 and the loop in line 6) take Θ(n1 + n2) = Θ(n) time. The last **for** loop (that is, the loop in line 12) makes n iterations, each taking constant time, for Θ(n) time. Therefore, the total running time is Θ(n).

**Analyzing Merge Sort**

For simplicity, assume that n is a power of 2 so that each divide step yields two subproblems, both of size exactly n/2.

The base case occurs when n = 1.

When n ≥ 2, time for merge sort steps:

* **Divide**: Just compute q as the average of p and r, which takes constant time i.e. Θ(1).
* **Conquer**: Recursively solve 2 subproblems, each of size n/2, which is 2T(n/2).
* **Combine**: MERGE on an n-element subarray takes Θ(n) time.

Summed together they give a function that is linear in n, which is Θ(n). Therefore, the recurrence for merge sort running time is

merge sort recurrence

**Solving the Merge Sort Recurrence**

By the master theorem in CLRS-Chapter 4 (page 73), we can show that this recurrence has the solution

T(n) = Θ(n lg n).

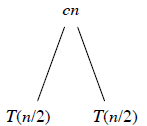
Reminder: lg n stands for log2 n.

Compared to insertion sort [Θ(n2) worst-case time], merge sort is faster. Trading a factor of n for a factor of lg n is a good deal. On small inputs, insertion sort may be faster. But for large enough inputs, merge sort will always be faster, because its running time grows more slowly than insertion sorts.

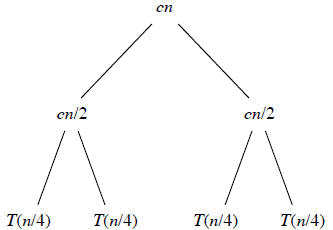
**Recursion Tree**

We can understand how to solve the merge-sort recurrence without the master theorem. There is a drawing of recursion tree on page 35 in CLRS, which shows successive expansions of the recurrence.

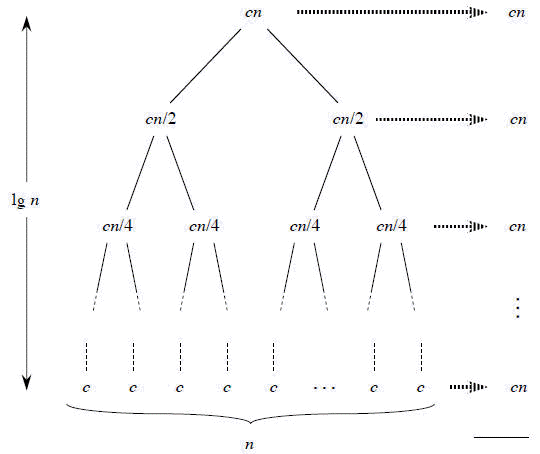
The following figure (Figure 2.5b in CLRS) shows that for the original problem, we have a cost of cn, plus the two subproblems, each costing T (n/2).



The following figure (Figure 2.5c in CLRS) shows that for each of the size-n/2 subproblems, we have a cost of cn/2, plus two subproblems, each costing T (n/4).



The following figure (Figure: 2.5d in CLRS) tells to continue expanding until the problem sizes get down to 1.



In the above recursion tree, each level has cost cn.

* The top level has cost cn.
* The next level down has 2 subproblems, each contributing cost cn/2.
* The next level has 4 subproblems, each contributing cost cn/4.
* Each time we go down one level, the number of subproblems doubles but the cost per subproblem halves. Therefore, cost per level stays the same.

The height of this recursion tree is lg n and there are lg n + 1 levels.

**Mathematical Induction**

We use induction on the size of a given subproblem n.

**Base case**: n = 1

Implies that there is 1 level, and lg 1 + 1 = 0 + 1 = 1.

**Inductive Step**

Our inductive hypothesis is that a tree for a problem size of 2i has lg 2i + 1 = i +1 levels. Because we assume that the problem size is a power of 2, the next problem size up after 2i is 2i + 1. A tree for a problem size of 2i + 1 has one more level than the size-2i tree implying i + 2 levels. Since lg 2i + 1 = i + 2, we are done with the inductive argument.

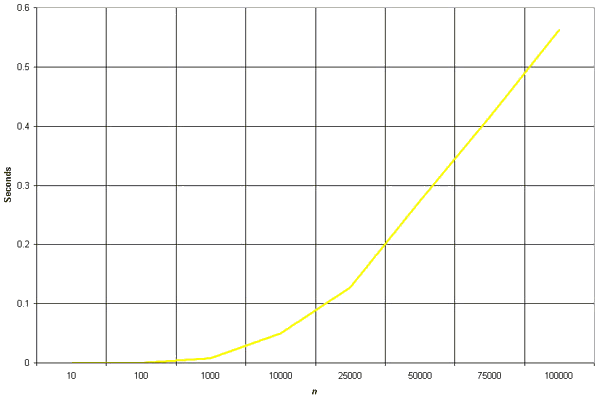
Total cost is sum of costs at each level of the tree. Since we have lg n +1 levels, each costing cn, the total cost is

cn lg n + cn.

Ignore low-order term of cn and constant coefÞcient c, and we have,

Θ(n lg *n*)

which is the desired result.



**Implementation**

void mergeSort(**int** numbers[], **int** temp[], **int** array\_size)

{  
        m\_sort(numbers, temp, 0, array\_size - 1);  
  
}  
  
  
  
void m\_sort(int numbers[], int temp[], int left, int right)  
  
{  
        int mid;  
  
        **if** (right > left)  
  
        {  
  
            mid = (right + left) / 2;  
  
            m\_sort(numbers, temp, left, mid);  
  
            m\_sort(numbers, temp, mid+1, right);  
  
  
            merge(numbers, temp, left, mid+1, right);  
  
        }  
  
}  
  
  
  
void merge(int numbers[], int temp[], int left, int mid, int right)  
  
        {  
  
            int i, left\_end, num\_elements, tmp\_pos;  
  
  
            left\_end = mid - 1;  
  
            tmp\_pos = left;  
  
            num\_elements = right - left + 1;  
  
  
  
**while** ((left <= left\_end) && (mid <= right))  
  
        {  
  
                **if** (numbers[left] <= numbers[mid])  
  
                {  
  
                        temp[tmp\_pos] = numbers[left];  
  
                        tmp\_pos = tmp\_pos + 1;  
  
                        left = left +1;  
  
                }  
  
                **else**  
  
                {  
  
                        temp[tmp\_pos] = numbers[mid];  
  
                        tmp\_pos = tmp\_pos + 1;  
  
                        mid = mid + 1;  
  
                }  
  
        }  
  
  
  
        **while** (left <= left\_end)  
  
                {  
  
                        temp[tmp\_pos] = numbers[left];  
  
                        left = left + 1;  
  
                        tmp\_pos = tmp\_pos + 1;  
  
                }  
  
                **while** (mid <= right)  
  
                {  
  
                        temp[tmp\_pos] = numbers[mid];  
  
                        mid = mid + 1;  
  
                        tmp\_pos = tmp\_pos + 1;  
  
                }  
  
  
  
                **for** (i = 0; i <= num\_elements; i++)  
  
                {  
  
                        numbers[right] = temp[right];  
  
                        right = right - 1;  
  
                }  
  
        }

|  |  |
| --- | --- |
| **Sorting algorithms**  **Mergesort** | [contents](http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/merge/mergenfra.htm)  [German version](http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/merge/merge.htm)  [up](http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/algoen.htm) |

The sorting algorithm Mergesort produces a sorted sequence by sorting its two halves and merging them. With a time complexity of *O*(*n* log(*n*)) Mergesort is optimal.

**Idea**

Similar to [Quicksort](http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/quick/quicken.htm), the Mergesort algorithm is based on a divide and conquer strategy[explanation](javascript:wind('../../../glossar/divconqe.htm',700,300,'no');). First, the sequence to be sorted is decomposed into two halves (*Divide*). Each half is sorted independently (*Conquer*). Then the two sorted halves are merged to a sorted sequence (*Combine*) (Figure 1).

|  |
| --- |
|  |
| |  | | --- | | Mergesort(n) | |
|  |
|  |
| Figure 1:  Mergesort(n) |
|  |

The following procedure *mergesort* sorts a sequence *a* from index *lo* to index *hi*.

|  |
| --- |
| void mergesort(int lo, int hi)  {  if (lo<hi)  {  int m=(lo+hi)/2;  mergesort(lo, m);  mergesort(m+1, hi);  merge(lo, m, hi);  }  } |

First, index *m* in the middle between *lo* and *hi* is determined. Then the first part of the sequence (from *lo* to *m*) and the second part (from *m*+1 to *hi*) are sorted by recursive calls of *mergesort*. Then the two sorted halves are merged by procedure *merge*. Recursion ends when *lo* = *hi*, i.e. when a subsequence consists of only one element.

The main work of the Mergesort algorithm is performed by function *merge*. There are different possibilities to implement this function.

**a) Straightforward variant of function *merge***

Function *merge* is usually implemented in the following way: The two halves are first copied into an auxiliary array *b*. Then the two halves are scanned by pointers *i* and *j* and the respective next-greatest element at each time is copied back to array *a* (Figure 2).

|  |
| --- |
|  |
| |  | | --- | | Merging two sorted halves | |
|  |
|  |
| Figure 2:  Merging two sorted halves |
|  |

At the end a situation occurs where one index has reached the end of its half, while the other has not. Then, in principle, the rest of the elements of the corresponding half have to be copied back. Actually, this is not necessary for the second half, since (copies of) the remaining elements are already at their proper places.

**Simulation (straightforward variant of function *merge*)**

The following program is an implementation of this approach.

|  |
| --- |
| // Straightforward variant  void merge(int lo, int m, int hi)  {  int i, j, k;  // copy both halves of a to auxiliary array b  for (i=lo; i<=hi; i++)  b[i]=a[i];  i=lo; j=m+1; k=lo;  // copy back next-greatest element at each time  while (i<=m && j<=hi)  if (b[i]<=b[j])  a[k++]=b[i++];  else  a[k++]=b[j++];  // copy back remaining elements of first half (if any)  while (i<=m)  a[k++]=b[i++];  } |

In Java the short form k++ is equivalent to k=k+1; the statement a[k++]=b[i++] is equivalent to the sequence a[k]=b[i]; k=k+1; i=i+1.

**b) Efficient variant of function *merge***

Actually, it is not necessary to copy the second half of array *a* to the auxiliary array *b*. It can remain where it is in array *a* (Fig. 3). In this way, only half as much auxiliary memory space is required, and also only half as much time to copy array elements to *b*. Furthermore, if all elements of the first half have been copied back to *a*, the remaining elements of the second half need not be moved anymore since they are already at their proper places [Som 04].

|  |
| --- |
|  |
| |  | | --- | | Merging b with second half of a | |
|  |
|  |
| Figure 3:  Merging *b* with second half of *a* |
|  |

Observe that when index *k* reaches index *j* all elements of array *b* have been copied back.

**Simulation (efficient variant of function *merge*)**

The implementation of this approach follows.

|  |
| --- |
| // Efficient variant  void merge(int lo, int m, int hi)  {  int i, j, k;  i=0; j=lo;  // copy first half of array a to auxiliary array b  while (j<=m)  b[i++]=a[j++];  i=0; k=lo;  // copy back next-greatest element at each time  while (k<j && j<=hi)  if (b[i]<=a[j])  a[k++]=b[i++];  else  a[k++]=a[j++];  // copy back remaining elements of first half (if any)  while (k<j)  a[k++]=b[i++];  } |

**c) Bitonic variant of function *merge***

In this approach, the first half of *a* is copied to auxiliary array *b* in its normal order, but the second half is copied to *b* in opposite order [Sed 88]. The result is a sequence that first monotonically increases and then monotonically decreases – a so-called *bitonic* sequence. Now this sequence is scanned by pointers *i* and *j* from both ends. Again, the respective next-greatest element at each time is copied back to array *a*. Copying is completed when *i* and *j* cross, i.e. when *i* > *j* (Figure 4). Observe that it is not necessary that *i* and *j* stay in "their" halves.

|  |
| --- |
|  |
| |  | | --- | | Merging of two halves sorted in opposite order | |
|  |
|  |
| Figure 4:  Merging of two halves sorted in opposite order |
|  |

**Simulation (bitonic variant of function *merge*)**

The implementation of the bitonic approach follows.

|  |
| --- |
| // Bitonic variant  void merge(int lo, int m, int hi)  {  int i=lo, j=hi, k=lo;  // copy first half of array a to auxiliary array b  while (i<=m)  b[k++]=a[i++];  // copy second half of array a to auxiliary array b,  // but in opposite order  while (j>m)  b[k++]=a[j--];  i=lo; j=hi; k=lo;  // copy back next-greatest element at each time  // until i and j cross  while (i<=j)  if (b[i]<=b[j])  a[k++]=b[i++];  else  a[k++]=b[j--];  } |

This variant of function *merge* requires in each case exactly the same number of steps – no matter if the input sequence is random, sorted or sorted in opposite direction.

In contrast to the other variants, it is not stable, i.e. it is possible that the original order of equal elements is changed.

**Program**

The following class *MergeSorter* encapsulates the functions *mergesort* and *merge*.

The method *sort* passes the array to be sorted to array *a*, allocates the auxiliary array *b* and calls *mergesort*.

The command for sorting an array *c* with mergesort is

|  |
| --- |
| MergeSorter.sort(c); |

The size of the auxiliary array *b* has to be chosen appropriately according to the implementation of function *merge*, namely *n* for variants a and c and (*n*+1)/2 for variant b.

|  |
| --- |
| public class MergeSorter  {  private static int[] a, b; // auxiliary array b  public static void sort(int[] a0)  {  a=a0;  int n=a.length;  // according to variant either/or:  b=new int[n]; b=new int[(n+1)/2];  mergesort(0, n-1);  }  private static void mergesort(int lo, int hi)  {  if (lo<hi)  {  int m=(lo+hi)/2;  mergesort(lo, m);  mergesort(m+1, hi);  merge(lo, m, hi);  }  }  private static void merge(int lo, int m, int hi)  {  // insert implementation here  }  } // end class MergeSorter |

**Analysis**

The straightforward version of function *merge* requires at most 2*n* steps (*n* steps for copying the sequence to the intermediate array *b*, and at most *n* steps for copying it back to array *a*). The [time complexity](http://www.iti.fh-flensburg.de/lang/algorithmen/asympen.htm) of *mergesort* is therefore

*T*(*n*) <= 2*n* + 2 *T*(*n*/2)   and

*T*(1)  =  0

The solution of this recursion yields

*T*(*n*) <= 2*n* log(*n*)  element  *O*(*n* log(*n*))

Thus, the mergesort algorithm is optimal, since the [lower bound](http://www.iti.fh-flensburg.de/lang/algorithmen/asympen.htm) for the sorting problem of Ω(*n* log(*n*)) is attained.

In the more efficient variant, function *merge* requires at most 1.5*n* steps (*n*/2 steps for copying the first half of the sequence to the intermediate array *b*, *n*/2 steps for copying it back to array *a*, and at most *n*/2 steps for processing the second half). This yields a running time of *mergesort* of at most 1.5*n* log(*n*) steps.

**Conclusions**

Algorithm mergesort has a time complexity of Θ(*n* log(*n*)) which is optimal. A drawback of mergesort is that it needs an additional space of Θ(*n*) for the temporary array *b*.

There are different possibilities to implement function *merge*. The most efficient of these is variant b. It requires only half as much additional space, it is faster than the other variants, and it is stable.

**References**

|  |  |  |
| --- | --- | --- |
|  |  |  |
| [Knu 73] | D.E. Knuth: The Art of Computer Programming, Vol. 3 - Sorting and Searching. Addison-Wesley (1973) | |
| [Sed 88] | R. Sedgewick: Algorithms. 2nd edition, Addison-Wesley (1988) | |
| [Som 04] | P. Sommerlad: (Persönliche Korrespondenz). Peter Sommerlad, Hochschule für Technik Rapperswil, Schweiz (2004) | |

|  |  |
| --- | --- |
| Next:  [[Shellsort]](http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/shell/shellen.htm)   or   [up](http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/algoen.htm) | [del.icio.us](http://del.icio.us/post/post?url=http://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/merge/mergen.htm&title=Mergesort)[digg.com](http://digg.com/submit?phase=2&url=http://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/merge/mergen.htm)[Google](http://www.google.com/bookmarks/mark?u=http://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/merge/mergen.htm&op=add&bkmk=http://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/merge/mergen.htm&title=Mergesort)[Ma.gnolia](http://ma.gnolia.com/beta/bookmarklet/add?url=http://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/merge/mergen.htm&title=Mergesort)[Mister Wong](http://www.mister-wong.de/index.php?action=addurl&bm_url=http://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/merge/mergen.htm&bm_description=Mergesort)[StumbleUpon](http://www.stumbleupon.com/submit?url=http://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/merge/mergen.htm&title=Mergesort)[YahooMyWeb](http://myweb.yahoo.com/myresults/bookmarklet?u=http://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/merge/mergen.htm&t=Mergesort)[LinkARENA](http://linkarena.com/bookmarks/addlink/?url=http://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/merge/mergen.htm&title=Mergesort) |

|  |  |
| --- | --- |
| [*[home](http://www.iti.fh-flensburg.de/lang/indexen.htm)H.W. Lang*](http://www.iti.fh-flensburg.de/lang/indexen.htm)[*FH Flensburg*](http://www.fh-flensburg.de)[*lang@fh-flensburg.de*](mailto:lang@fh-flensburg.de)[*Impressum*](http://www.iti.fh-flensburg.de/lang/impressum.htm) *©   Created: 19.01.2000   Updated: 18.05.2010* | [Valid HTML 4.01 Transitional](http://validator.w3.org/check?uri=referer) |