Merge Sort

Like [QuickSort](https://www.geeksforgeeks.org/quick-sort/), Merge Sort is a [Divide and Conquer](https://www.geeksforgeeks.org/divide-and-conquer-introduction/) algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. **The merge() function** is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one. See following C implementation for details.

**MergeSort(arr[], l, r)**

If r > l

**1.** Find the middle point to divide the array into two halves:

middle m = (l+r)/2

**2.** Call mergeSort for first half:

Call mergeSort(arr, l, m)

**3.** Call mergeSort for second half:

Call mergeSort(arr, m+1, r)

**4.** Merge the two halves sorted in step 2 and 3:

Call merge(arr, l, m, r)

The following diagram from [wikipedia](http://en.wikipedia.org/wiki/File:Merge_sort_algorithm_diagram.svg" \t "_blank) shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}. If we take a closer look at the diagram, we can see that the array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge processes comes into action and starts merging arrays back till the complete array is merged.



[**Recommended: Please solve it on “*PRACTICE*” first, before moving on to the solution.**](https://practice.geeksforgeeks.org/problems/merge-sort/1)

|  |
| --- |
| /\* Java program for Merge Sort \*/  class MergeSort  {      // Merges two subarrays of arr[].      // First subarray is arr[l..m]      // Second subarray is arr[m+1..r]      void merge(int arr[], int l, int m, int r)      {          // Find sizes of two subarrays to be merged          int n1 = m - l + 1;          int n2 = r - m;            /\* Create temp arrays \*/          int L[] = new int [n1];          int R[] = new int [n2];            /\*Copy data to temp arrays\*/          for (int i=0; i<n1; ++i)              L[i] = arr[l + i];          for (int j=0; j<n2; ++j)              R[j] = arr[m + 1+ j];              /\* Merge the temp arrays \*/            // Initial indexes of first and second subarrays          int i = 0, j = 0;            // Initial index of merged subarry array          int k = l;          while (i < n1 && j < n2)          {              if (L[i] <= R[j])              {                  arr[k] = L[i];                  i++;              }              else              {                  arr[k] = R[j];                  j++;              }              k++;          }            /\* Copy remaining elements of L[] if any \*/          while (i < n1)          {              arr[k] = L[i];              i++;              k++;          }            /\* Copy remaining elements of R[] if any \*/          while (j < n2)          {              arr[k] = R[j];              j++;              k++;          }      }        // Main function that sorts arr[l..r] using      // merge()      void sort(int arr[], int l, int r)      {          if (l < r)          {              // Find the middle point              int m = (l+r)/2;                // Sort first and second halves              sort(arr, l, m);              sort(arr , m+1, r);                // Merge the sorted halves              merge(arr, l, m, r);          }      }        /\* A utility function to print array of size n \*/      static void printArray(int arr[])      {          int n = arr.length;          for (int i=0; i<n; ++i)              System.out.print(arr[i] + " ");          System.out.println();      }        // Driver method      public static void main(String args[])      {          int arr[] = {12, 11, 13, 5, 6, 7};            System.out.println("Given Array");          printArray(arr);            MergeSort ob = new MergeSort();          ob.sort(arr, 0, arr.length-1);            System.out.println("\nSorted array");          printArray(arr);      }  }  /\* This code is contributed by Rajat Mishra \*/ |

**Output:**

Given array is

12 11 13 5 6 7

Sorted array is

5 6 7 11 12 13

**Time Complexity:** Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.  
T(n) = 2T(n/2) +   
The above recurrence can be solved either using Recurrence Tree method or Master method. It falls in case II of Master Method and solution of the recurrence is .  
Time complexity of Merge Sort is  in all 3 cases (worst, average and best) as merge sort always divides the array into two halves and take linear time to merge two halves.

**Auxiliary Space:** O(n)

**Algorithmic Paradigm:**Divide and Conquer

**Sorting In Place:** No in a typical implementation

**Stable:** Yes

**Applications of Merge Sort**

1. [Merge Sort is useful for sorting linked lists in O(nLogn) time](https://www.geeksforgeeks.org/merge-sort-for-linked-list/).In the case of linked lists, the case is different mainly due to the difference in memory allocation of arrays and linked lists. Unlike arrays, linked list nodes may not be adjacent in memory. Unlike an array, in the linked list, we can insert items in the middle in O(1) extra space and O(1) time. Therefore merge operation of merge sort can be implemented without extra space for linked lists.

In arrays, we can do random access as elements are contiguous in memory. Let us say we have an integer (4-byte) array A and let the address of A[0] be x then to access A[i], we can directly access the memory at (x + i\*4). Unlike arrays, we can not do random access in the linked list. Quick Sort requires a lot of this kind of access. In linked list to access i’th index, we have to travel each and every node from the head to i’th node as we don’t have a continuous block of memory. Therefore, the overhead increases for quicksort. Merge sort accesses data sequentially and the need of random access is low.

1. [Inversion Count Problem](https://www.geeksforgeeks.org/counting-inversions/)
2. Used in [External Sorting](http://en.wikipedia.org/wiki/External_sorting)

Merge sort is one of the most efficient sorting algorithms. It works on the principle of Divide and Conquer. Merge sort repeatedly breaks down a list into several sublists until each sublist consists of a single element and merging those sublists in a manner that results into a sorted list.

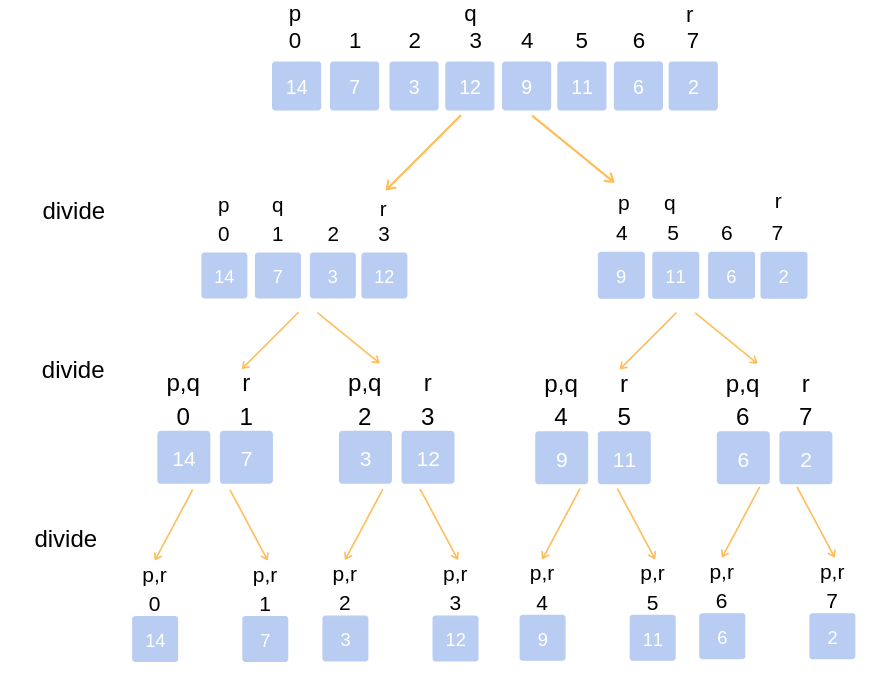
## A merge sort works as follows:

### Top-down Merge Sort Implementation:

The top-down merge sort approach is the methodology which uses recursion mechanism. It starts at the Top and proceeds downwards, with each recursive turn asking the same question such as “What is required to be done to sort the array?” and having the answer as, “split the array into two, make a recursive call, and merge the results.”, until one gets to the bottom of the array-tree.

Example: Let us consider an example to understand the approach better.

1. Divide the unsorted list into n sublists, each comprising 1 element (a list of 1 element is supposed sorted).

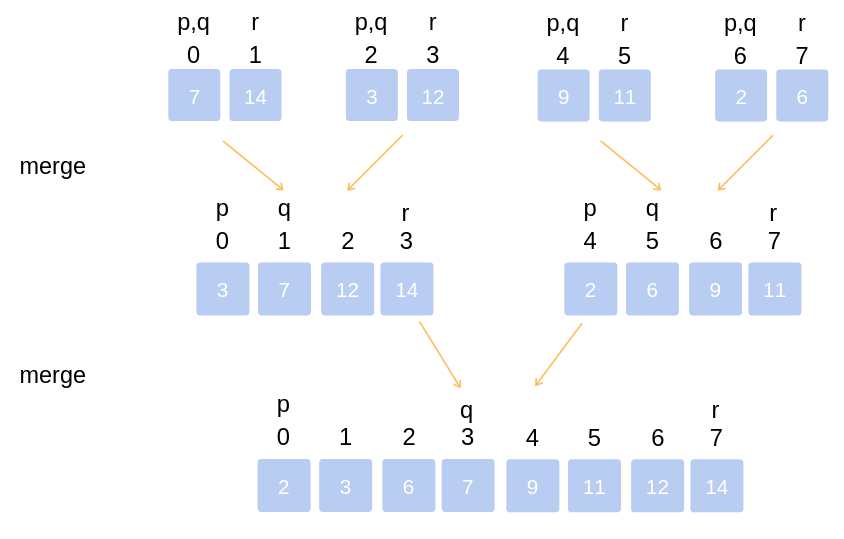


Top-down Implementation

1. Repeatedly merge sublists to produce newly sorted sublists until there is only 1 sublist remaining. This will be the sorted list.

### Merging of two lists done as follows:

The first element of both lists is compared. If sorting in ascending order, the smaller element among two becomes a new element of the sorted list. This procedure is repeated until both the smaller sublists are empty and the newly combined sublist covers all the elements of both the sublists.

****

Merging of two lists

## Implementation Of Merge Sort

*// example of merge sort in Java*

*// merge function take two intervals*

*// one from start to mid*

*// second from mid+1, to end*

*// and merge them in sorted order*

**void** **merge(int** Arr**[],** **int** start**,** **int** mid**,** **int** end**)** **{**

*// create a temp array*

**int** temp**[]** **=** **new** **int[**end **-** start **+** 1**];**

*// crawlers for both intervals and for temp*

**int** i **=** start**,** j **=** mid**+**1**,** k **=** 0**;**

*// traverse both arrays and in each iteration add smaller of both elements in temp*

**while(**i **<=** mid **&&** j **<=** end**)** **{**

**if(**Arr**[**i**]** **<=** Arr**[**j**])** **{**

temp**[**k**]** **=** Arr**[**i**];**

k **+=** 1**;** i **+=** 1**;**

**}**

**else** **{**

temp**[**k**]** **=** Arr**[**j**];**

k **+=** 1**;** j **+=** 1**;**

**}**

**}**

*// add elements left in the first interval*

**while(**i **<=** mid**)** **{**

temp**[**k**]** **=** Arr**[**i**];**

k **+=** 1**;** i **+=** 1**;**

**}**

*// add elements left in the second interval*

**while(**j **<=** end**)** **{**

temp**[**k**]** **=** Arr**[**j**];**

k **+=** 1**;** j **+=** 1**;**

**}**

*// copy temp to original interval*

**for(**i **=** start**;** i **<=** end**;** i **+=** 1**)** **{**

Arr**[**i**]** **=** temp**[**i **-** start**]**

**}**

**}**

*// Arr is an array of integer type*

*// start and end are the starting and ending index of current interval of Arr*

**void** **mergeSort(int** Arr**[],** **int** start**,** **int** end**)** **{**

**if(**start **<** end**)** **{**

**int** mid **=** **(**start **+** end**)** **/** 2**;**

mergeSort**(**Arr**,** start**,** mid**);**

mergeSort**(**Arr**,** mid**+**1**,** end**);**

merge**(**Arr**,** start**,** mid**,** end**);**

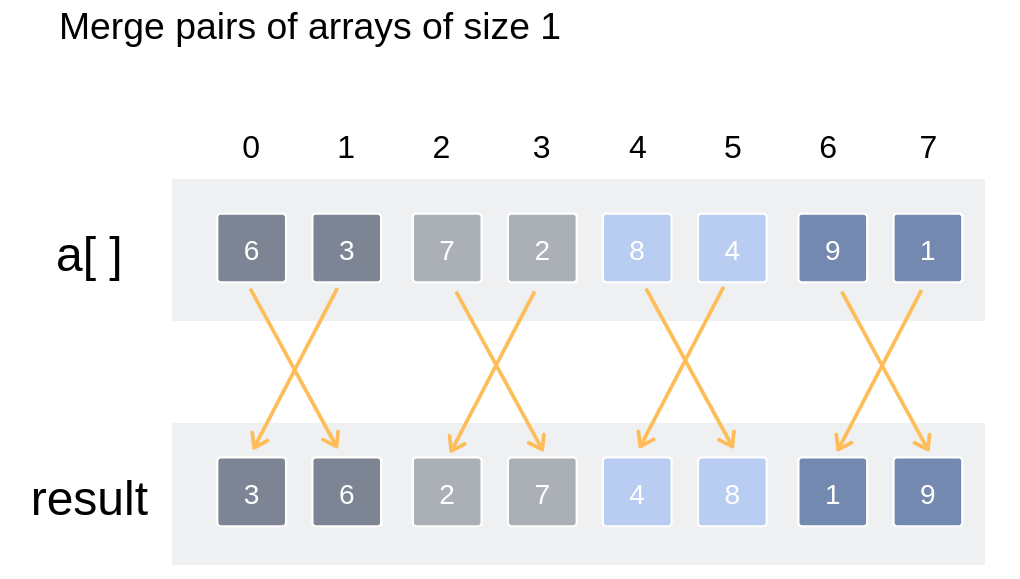
**}**

**}**

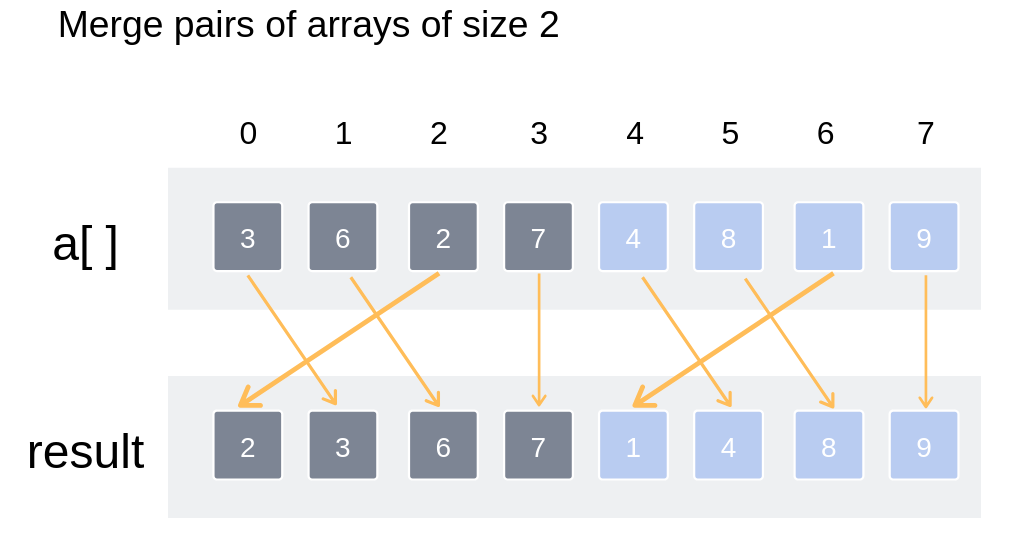
### Bottom-Up Merge Sort Implementation:

The Bottom-Up merge sort approach uses iterative methodology. It starts with the “single-element” array, and combines two adjacent elements and also sorting the two at the same time. The combined-sorted arrays are again combined and sorted with each other until one single unit of sorted array is achieved.

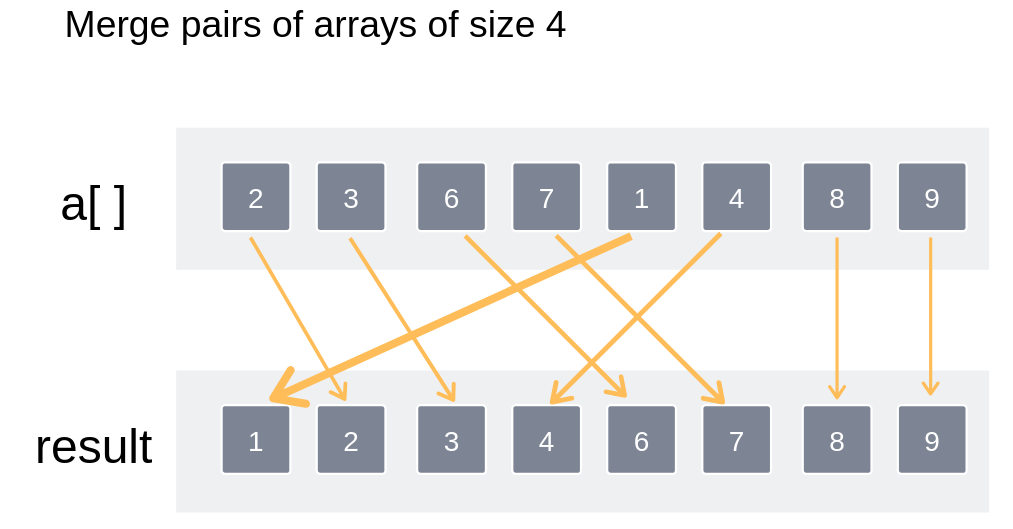
Example: Let us understand the concept with the following example.

1. Iteration (1)  
   

Merge pairs of arrays of size 1

1. Iteration (2)  
   ****

Merge pairs of arrays of size 2

1. Iteration (3)  
   

Merge pairs of arrays of size 4

Thus the entire array has been sorted and merged.