**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.

**Step by Step Process**

The merge sort algorithm is performed using following steps...

**Step 1:** Divide the unsorted list into n sublists, each comprising 1 element (a list of 1 element is

supposed sorted).

**Step 2:** Repeatedly merge sublists to produce newly sorted sublists until there is only 1 sublist

remaining. This will be the sorted list.

**Sorting Logic**

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.

Following is the sample code for merge sort...

**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 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.

Example



**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 in 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 case of linked lists the case is different mainly due to difference in memory allocation of arrays and linked lists. Unlike arrays, linked list nodes may not be adjacent in memory. Unlike array, in 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 continuous 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 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 continuous block of memory. Therefore, the overhead increases for quick sort. Merge sort accesses data sequentially and the need of random access is low.

1. Inversion Count Problem
2. Used in External Sorting

**Merge Sort Program in C Programming Language**

#include<stdio.h>

#include<stdlib.h>

void Merge(int a[], int tmp[], int lpos, int rpos, int rend)

{

int i, lend, n, tmppos;

lend = rpos - 1;

tmppos = lpos;

n = rend - lpos + 1;

while(lpos <= lend && rpos <= rend)

{

if(a[lpos] <= a[rpos])

tmp[tmppos++] = a[lpos++];

else

tmp[tmppos++] = a[rpos++];

}

while(lpos <= lend)

tmp[tmppos++] = a[lpos++];

while(rpos <= rend)

tmp[tmppos++] = a[rpos++];

for(i = 0; i < n; i++, rend--)

a[rend] = tmp[rend];

}

void MSort(int a[], int tmp[], int left, int right)

{

int center;

if(left < right)

{

center = (left + right) / 2;

MSort(a, tmp, left, center);

MSort(a, tmp, center + 1, right);

Merge(a, tmp, left, center + 1, right);

}

}

void MergeSort(int a[], int n)

{

int \*tmparray;

tmparray = malloc(sizeof(int) \* n);

MSort(a, tmparray, 0, n-1);

free(tmparray);

}

main()

{

int i, n, a[10];

printf("Enter the number of elements :: ");

scanf("%d",&n);

printf("Enter the elements :: ");

for(i = 0; i < n; i++)

{

scanf("%d",&a[i]);

}

MergeSort(a,n);

printf("The sorted elements are :: ");

for(i = 0; i < n; i++)

printf("%d ",a[i]);

printf("\n");

}

**OUTPUT:**

Enter the number of elements :: 7

Enter the elements :: 70 60 50 40 10 20 30

The sorted elements are :: 10 20 30 40 50 60 70