# Merge Sort

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. Using the Divide and Conquer technique, we divide a problem into subproblems. When the solution to each subproblem is ready, we 'combine' the results from the subproblems to solve the main problem.

**Divide and Conquer Strategy**

Suppose we had to sort an array A. A subproblem would be to sort a sub-section of this array starting at index p and ending at index r, denoted as A[p..r].

**Divide**

If q is the half-way point between p and r, then we can split the subarray A[p..r] into two arrays A[p..q] and A[q+1, r].

**Conquer**

In the conquer step, we try to sort both the subarrays A[p..q] and A[q+1, r]. If we haven't yet reached the base case, we again divide both these subarrays and try to sort them.

**Combine**

When the conquer step reaches the base step and we get two sorted subarrays A[p..q] and A[q+1, r] for array A[p..r], we combine the results by creating a sorted array A[p..r] from two sorted subarrays A[p..q] and A[q+1, r].

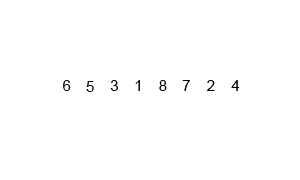
**The MergeSort Algorithm**

The MergeSort function repeatedly divides the array into two halves until we reach a stage where size of thr subarray of size 1 i.e. p == r.

After that, the merge function comes into play and combines the sorted arrays into larger arrays until the whole array is merged.

To sort an entire array, we need to call MergeSort(A, 0, length(A)-1).

First divide the list into the smallest unit (1 element), then compare each element with the adjacent list to sort and merge the two adjacent lists. Finally all the elements are sorted and merged.

[](file:///C:\Users\hp\Desktop\Merge+Sort+Animation.gif)



1. Main() =>mergeSort(A,0,6)

mergeSort(A,0,3) mergeSort(A,4,6) merge(A,0,3,6)

mergeSort(A,0,1) mergeSort(A,2,3) merge(A,0,1,3) merge(A,2,2,3)

mergeSort(A,0,0) mergeSort(A,1,1) merge(A,0,0,1) mergeSort(A,2,2) mergeSort(A,3,3)

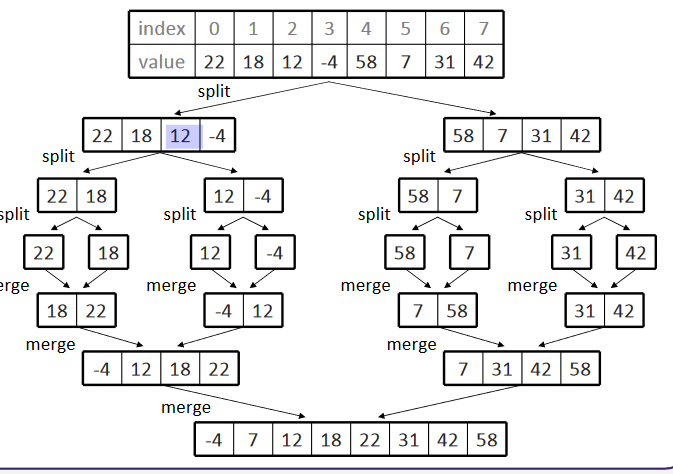
merge(A,4,6)

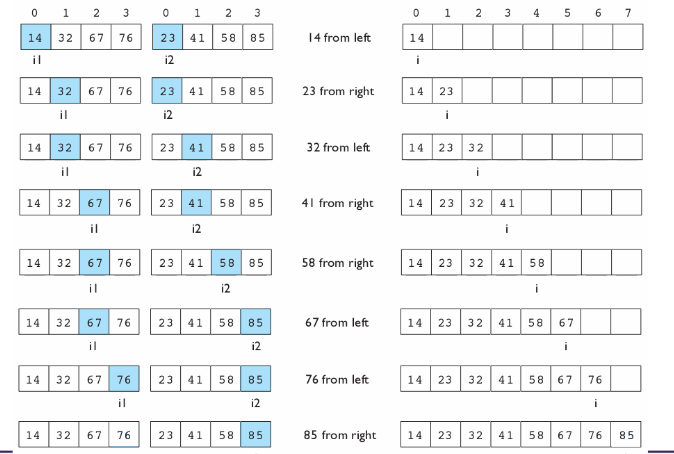
mergeSort(A,4,5) mergeSort(A,6,6) merge(A,4,5,6)

mergeSort(A,6,6) mergeSort(A,7,6) merge(A,6,6,6)

mergeSort(A,4,4) mergeSort(A,5,5) merge(A,4,4,5)







// Merges two subarrays of arr[]. // First subarray is arr[l..m] // Second subarray is arr[m+1..r]

void merge(int a[], int low, int mid, int high)

{

int l1, l2, i = low, b[100];

for(l1 = low, l2 = mid + 1,; l1 <= mid && l2 <= high;)

{

if(a[l1] <= a[l2])

b[i++] = a[l1++];

else

b[i++] = a[l2++];

}

while(l1 <= mid)

b[i++] = a[l1++];

while(l2 <= high)

b[i++] = a[l2++];

for(i = low; i <= high; i++)

a[i] = b[i];

}

/\* l is for left index and r is right index of the sub-array of arr to be sorted \*/

void mergeSort(int arr[], int l, int r)

{

if (l < r)

{

// Same as (l+r)/2, but avoids overflow for // large l and h

int m = (l+r)/2;

// Sort first and second halves

mergeSort(arr, l, m);

mergeSort(arr, m+1, r);

merge(arr, l, m, r);

}

}

/\* Driver program to test above functions \*/

int main()

{

int arr[100],n,i;

printf("Enter Element of the array");

scanf("%d",&n)

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

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

mergeSort(arr, 0, n - 1);

printf("\nSorted array is \n");

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

printf("%d\t",arr[i]);

return 0;

}

Worst Case Time Complexity [ Big-O ]: **O(n\*log n)**

Best Case Time Complexity [Big-omega]: **O(n\*log n)**

Average Time Complexity [Big-theta]: **O(n\*log n)**