**Experiment No: 2 Date:- 02-04-2021**

**AIM:** Implementation of Merge Sort Using Divide and Conquer and obtaining its step count

**THEORY:**

Merge Sort is a Divide and Conquer algorithm. It divides the input array into 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, I, m, r) is a key process that assumes that arr[...m] and arr[m+1.r] are sorted and merges the two sorted sub-arrays into one.

Merge sort runs in O(nlogn) time in all the cases.

**ALGORITHM:**

Algorithm MergeSort(low, high)

//a[low: high] is a global array to be sorted. Small(P) is true if there is only one element to sort. /In this case the list is already sorted

{

If ( low < high ) then //if there are more than

one element

{

//divide P into subproblems. Find

where to split the set

mid := [ (low+high)/2]; //solve the subproblems.

MergeSort (low, mid); Merge Sort(mid+1, high); //combine the solutions

Merge(low,mid,high);

}

}

Algorithm Merge(low,mid,high)

//a[low:high] is a global array to containing two sorted subsets in a a[low:mid] and in a[mid+1:high]. //the goal is to merge these two sets into a single set residing in a[low:high].

//b[ is an auxiliary global array

{

h:=low;I :=low; j := mid+1;

while ((h<mid) and (j<high)) do

{

If (a[h]<a[j]) then

{

b[i]:= a[h]; h:=h1;

}

else

{

b[i]:= a[j]; j:=j1;

}

i := i+1;

}

If (h>mid) then

for k := j to high do

{

b[i]:=a[k];i:=i+1;

}

Else

for k:=h to mid do

{

b[i]:=a[k];

i:=i+1

}

for k:=low to high do a[k] := b[k];

}

**Recurrence Relation Formulation and Solving:**

Time Complexity: Merge Sort is a recursive algorithm, and its time complexity can be expressed as following recurrence relation.

T(n) = 2T(n/2) + cn

The above recurrence can be solved either using the Recurrence Tree method or the Master method. It falls in case II of Master Method and the solution of the recurrence is O(nLogn).

Master's theorem a=2 b=2 f(n)=cn

n log (base b)

n(log, (2)

n'=n

T(n) =n\*U(n)

u(n) => h(n) = f(n)/n=cn/n=c=>r=0=>i=0

(Log(n))(i+1)/(i+1)

=log(n)

T(n) = n\*log(n) = O(nlog(n))

Time complexity of Merge Sort is O(nLogn) in all 3 cases (worst, average and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.

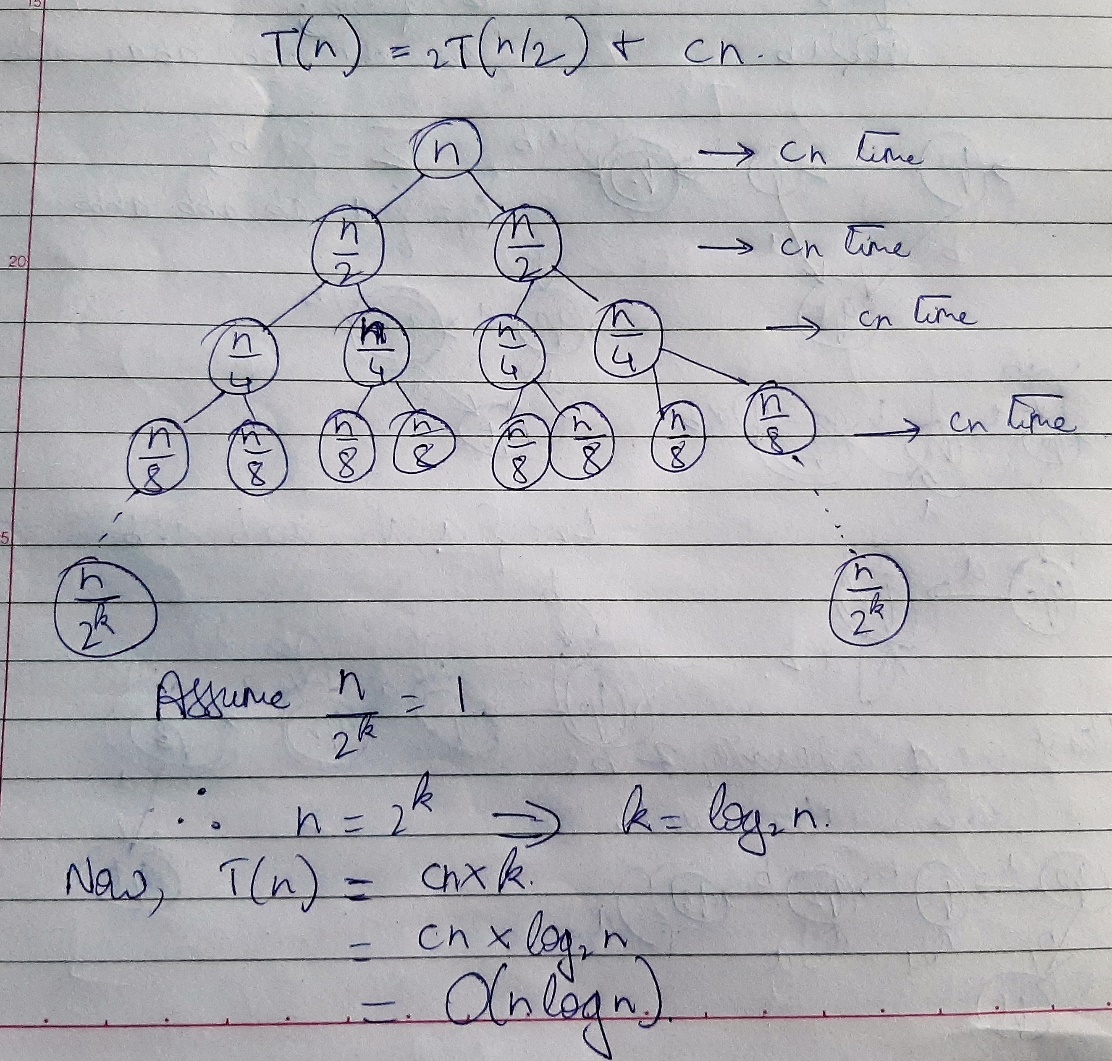
**Auxiliary space needed**: O(n) for the temporary array.

**Algorithm Paradigm**: Divide and Conquer

**Stability:** Yes

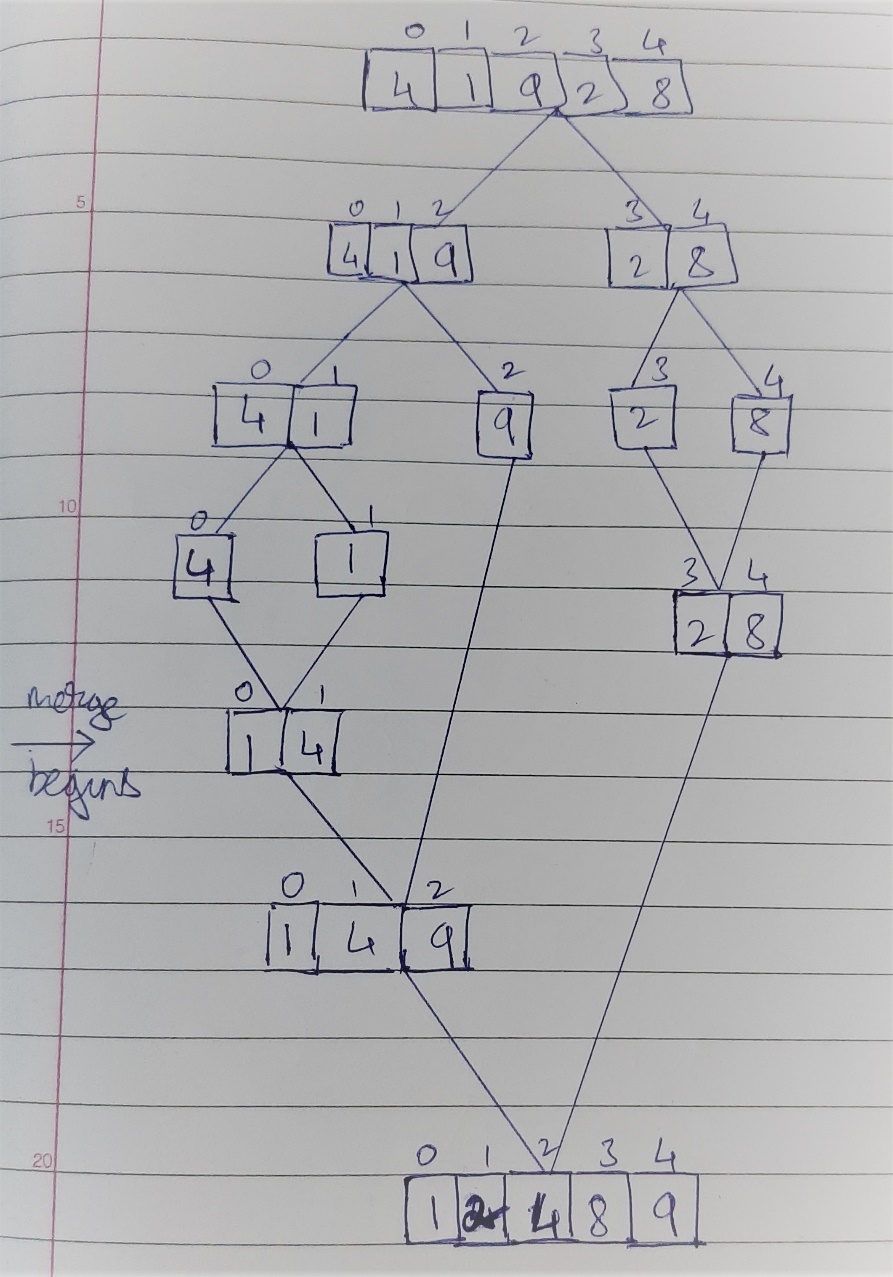
**Recursive Tree call Diagram:**

Shown below, is the recursive tree diagram:



**Problem Tracing:**

Consider array a={4,1,9,2,8 }. Here size of array n=5.



PROGRAM IMPLEMENTATION:

#include<iostream>

using namespace std;

void mergesort(int,int);

void merge(int,int,int);

int count=0;

int \*a,\*b;

int main()

{

int n;

cout<<"Enter number of elements\n";

count++;

cin>>n;

count++;

a = new int[n];

b = new int[n];

cout<<"Enter "<<n<<" elements to be sorted\n";

count++;

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

{

count++;//for

cin>>a[i];

count++;

}

mergesort(0,n-1);

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

cout<<a[i]<<" ";

cout<<endl;

cout<<"Count="<<count<<endl;

return 0;

}

void mergesort(int low, int high)

{

count++;

if(low<high)

{

int mid = (low+high)/2;

count++;

mergesort(low,mid);

mergesort(mid+1,high);

merge(low,mid,high);

}

}

void merge(int low, int mid, int high)

{

int i=low,j=mid+1,k=low;

while(i<=mid && j<=high)

{

count++; //while

count++; //if

if(a[i]<a[j])

{

count+=3;

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

}

else

{

b[k++] = a[j++];

count+=3;

}

}

count++; //while

while(i<=mid)

{

count++; //while statement

count++;

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

}

count++; //Last while

while(j<=high)

{

count++; //while statement

count++; //for

b[k++] = a[j++];

}

count++; //Last while

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

{

count++; //for

a[k]=b[k];

count++;

}

}

OUTPUTS:

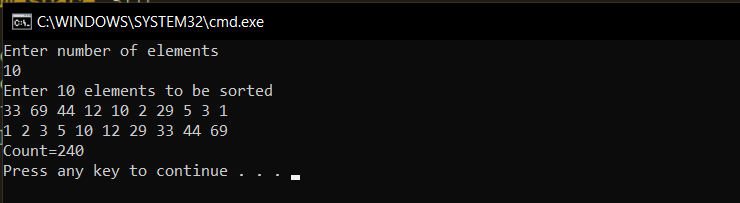
1. When elements are already sorted

**Count=238**



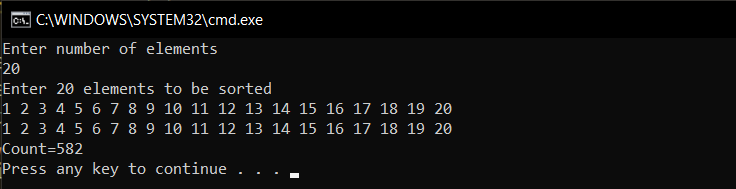
1. When elements are in random order

**Count=240**



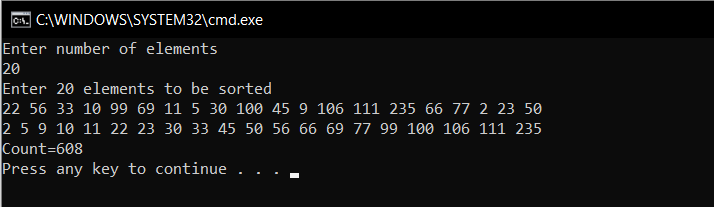
1. When n=20 and elements are in sorted order

**Count=582**

****

1. When n=20 and elements are in random order

**Count=608**

****

**Conclusion**: The program was successfully run and executed with main emphasis on the complexities of MergeSort Algorithm. The following was also observed.

1. The time complexity of Mergesort algorithm for all cases. I.e. best, average and worst case is O(nlogn).
2. The algorithm requires additional memory space O(n) for the temporary array.