Sample Experiment

1 AIM: Implement Merge Sort .

2 TOOLS/APPARATUS: Turbo C or gcc / gprof compiler in linux.

3 STANDARD PROCEDURES:

3.1 Analyzing the Problem:

Using divide and Conquer approach in merge sort method, we have to sort n number of data.

For Example: If we have input data like

38 27 43 3 9 82 10

Stepwise solution will be:

38 27 43 3 | 9 82 10
38 27 | 43 3 | 9 82 | 10
38 | 27 | 43 |3 |9 |82 | 10
27 38 | 3 43 | 9 82 | 10
3 27 38 43 | 9 10 82
3 9 10 27 38 43 82

3.2 Designing the Solution:

Algorithm of Merge Sort:

```
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 then one element
    {
        //Divide P into subproblems.
            //Find where to split the set.
            Mid:=(low+high)/2;
        //Solve the subproblems.
        Mergesort(low,mid);
        Mergesort(mid+1,high);
        //Combine the solution.
        Merge(low,mid,high);
    }
}
Algorithm Merge(low,mid,high)
//a[low:high] is a global array containing two sorted subsets in a[low:mid]
// and in a[mid+1:high].The goal is to merge these two serts 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:=h+1;
                }
                Else
                {
                        b[i]:=a[j];j:=j+1;
                }
                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];
}
```

3.3 Implementing the Solution

3.3.1 Writing Source Code:

```
#include<iostream.h>
#include<conio.h>
void merge(int *,int,int,int);//shows how data will be
divided,getting sorted and merged.
void ms(int *,int ,int );//Divide data into subparts and merge them
void main()
{    clrscr();
        int n, a[10], i;//a[]will store input.
        cout<<"Enter the number of elemets =";
        cin>>n;
        cout<<"Enter the elements of arry for sorting = ";
        for(i=0;i<n;i++)
        {
                cin>>a[i];
        }
        ms(a,0,n-1);
        cout<<"Sorted elements : ";
        for(i=0;i<n;i++)
        cout<<a[i]<<' ';
    getch();
}
void ms(int *a,int low,int high)
{
```

```
        //low indicates index of first data and high indicates index
    of last data.
        if(low<high)
        {
            int mid=(low+high)/2;
            ms(a,low,mid);
            ms(a,mid+1,high);
            merge(a,low,mid,high);
        }
    }
    void merge(int *a, int low, int mid,int high)
    {
        int temp[10];
        int h=low,i=low,j=mid+1;
        while(i<=mid&&j<=high)
        {
            if(a[i]<=a[j])
            {
                temp[h++]=a[i++];
            }
            else
            temp[h++]=a[j++];
        }
        if(i>mid)
        {
            for(int k=j;k<=high;k++)
            temp[h++]=a[k];
        }
        else
        {
            for(int k=i;k<=mid;k++)
            temp[h++]=a[k];
        }
        for(int k=low;k<=high;k++)
        {
            a[k]=temp[k];
        }
    }
```

3.3.2 Compilation /Running and Debugging the Solution

In linux,

```
Gcc mergesort.c
./a.out
Enter the number of elemets =5
Enter the elements of arry for sorting = 2 3 1 5 4
Sorted elements :1 2 3 4 5
```

3.4 Testing the Solution

By giving command gprof a.out > abc
vi abc
output will be:

Flat profile:

Each sample counts as 0.01 seconds.
no time accumulated
% cumulative self
self total
time seconds seconds calls Ts/call Ts/call name
0.00 0.00 0.00 1 0.00 0.00 frame_dummy
% the percentage of the total running time of the
time program used by this function.
cumulative a running sum of the number of seconds accounted
seconds for by this function and those listed above it.
self the number of seconds accounted for by this seconds function alone.This is the major sort for this listing.
calls the number of times this function was invoked, if this function is profiled, else blank.
self the average number of milliseconds spent in this
ms/call function per call, if this function is profiled ,else blank.
total the average number of milliseconds spent in this
ms/call function and its descendents per call, if this function is profiled, else blank.
name the name of the function. This is the minor sort for this listing. The index shows the location of the function in the gprof listing. If the index is
in parenthesis it shows where it would appear in the gprof listing if it were to be printed.

Call graph (explanation follows)
granularity: each sample hit covers 4 byte(s) no time propagated
index % time self children called name
0.00 0.00 1/1 __do_global_dtors_aux [11]
[1] 0.0 0.00 0.00 1 frame_dummy [1]

This table describes the call tree of the program, and was sorted by the total amount of time spent in each function and its children.
Each entry in this table consists of several lines. The line with the
index number at the left hand margin lists the current function.
The lines above it list the functions that called this function,
and the lines below it list the functions this one called.
This line lists:
index A unique number given to each element of the table.

Index numbers are sorted numerically.
The index number is printed next to every function name so it is easier to look up where the function in the table.

% time This is the percentage of the `total' time that was spent in this function and its children. Note that due to different viewpoints, functions excluded by options, etc, these numbers will NOT add up to 100%.

self This is the total amount of time spent in this function.

children This is the total amount of time propagated into this function by its children.

called This is the number of times the function was called. If the function called itself recursively, the number only includes non-recursive calls, and is followed by a `+' and the number of recursive calls.

name The name of the current function. The index number is printed after it. If the function is a member of a cycle, the cycle number is printed between the function's name and the index number.

For the function's parents, the fields have the following meanings:

self This is the amount of time that was propagated directly from the function into this parent.

children This is the amount of time that was propagated from the function's children into this parent.

called This is the number of times this parent called the function `/' the total number of times the function was called. Recursive calls to the function are not included in the number after the `/'.

name This is the name of the parent. The parent's index number is printed after it. If the parent is a member of a cycle, the cycle number is printed between the name and the index number.

If the parents of the function cannot be determined, the word `<spontaneous>' is printed in the `name' field, and all the other fields are blank. For the function's children, the fields have the following meanings:

self This is the amount of time that was propagated directly from the child into the function.

children This is the amount of time that was propagated from the child's children to the function.

called This is the number of times the function called this child `/' the total number of times the child was called. Recursive calls by the child are not listed in the number after the `/'.

name This is the name of the child. The child's index number is printed after it. If the child is a member of a cycle, the cycle number is printed between the name and the index number.

If there are any cycles (circles) in the call graph, there is an

entry for the cycle-as-a-whole. This entry shows who called the
cycle (as parents) and the members of the cycle (as children.)
The `+' recursive calls entry shows the number of function calls that
were internal to the cycle, and the calls entry for each member shows,
for that member, how many times it was called from other members of
the cycle.Index by function name
[1] frame_dummy

4 Conclusions

Time Complexity Of Merge Sort in best case is(when all data is already in sorted form):O(n) Time Complexity Of Merge Sort in worst case is: O(n logn)
Time Complexity Of Merge Sort in average case is: O(n logn)

EXPERIMENT - 2

Aim: a) Write a program to implement Tower of Hanoii problem.
Procedure:

- Move n number of disks from tower a to tower b.
- Use tower c as intermediate tower.

b) Show the solution for 3 disk 3 pan problem. How many optimal step is needed.