## Experiment No: 3

***Title:*** Implementation of Merge Sort algorithm using Divide & Conquer method

##### Theory/Description:

***Divide-and-Conquer Algorithms:*** These algorithms have the following outline: To solve a problem, divide it into sub problems. Recursively solve the sub problems. Finally glue the resulting solutions together to obtain the solution to the original problem. Progress here is measured by how much smaller the sub problems are compared to the original problem.

You have already seen an example of a divide and conquer algorithm in 3 rd assignment. First we will see MERGESORT.

The idea behind merge sort is to take a list, divide it into two smaller sub lists, conquer each sub list by sorting it, and then combine the two solutions for the sub problems into a single solution. These three basic steps {divide,conquer,andcombine {lie behind most divide and conquer algorithms.

With merge sort, we kept dividing the list into halves until there was just one element left. In general, we may divide the problem into smaller problems in any convenient fashion. Also, in practice it may not be best to keep dividing until the instances are completely trivial. Instead, it may be wise to divide until the instances are reasonably small, and then apply an algorithm that is faster on small instances.

Merge-sort is based on the divide-and-conquer paradigm. The Merge-sort algorithm can be described in general terms as consisting of the following three steps:

* + 1. Divide Step

If given array A has zero or one element, return S; it is already sorted. Otherwise, divide A into two arrays, A1 and A2, each containing about half of the elements of A.

* + 1. Recursion Step

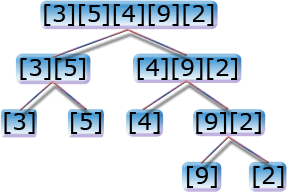
Recursively sort array A1 and A2.

* + 1. Conquer Step

Combine the elements back in A by merging the sorted arrays A1 and A2 into a sorted sequence.

We can visualize Merge-sort by means of binary tree where each node of the tree represents

a recursive call and each external node represent individual elements of given array A. Such a tree is called Merge-sort tree. The heart of the Merge-sort algorithm is conquer step, which merge two sorted sequences into a single sorted sequence.



To begin, suppose that we have two sorted arrays A1[1], A1[2], . . , A1[M] and A2[1], A2[2], . . . , A2[N]. The following is a direct algorithm of the obvious strategy of successively choosing the smallest remaining elements from A1 to A2 and putting it in A.

Merging two sorted subarrays:

**Algorithm Merge (low, high, mid)**

// a [low: high] is a global array containing two sorted subsets ina [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

{

i=i+1;

}

}

else

{

}

b[i]=a[h]; h=h+1;

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

}

Merge Sort

Algorithm MergeSort(low,high)

// a [low: high] is a global array to be sorted.

{

if(low<high) then

{

// divide array into subarrays ; mid=[(low+high)/2]; MergeSort(low,mid); MergeSort(mid+1,high);

// combine the sorted subarrays Merge(low,mid,high);

}

}

Program Snippets:

void mergeSort(int numbers[], int temp[], intarray\_size)

{

m\_sort(numbers, temp, 0, array\_size - 1);

}

voidm\_sort(int numbers[], int temp[], int left, int right)

{

intmid;

if (right > left)

{

mid= (right + left) / 2; m\_sort(numbers, temp, left, mid); m\_sort(numbers, temp, mid+1,right);

merge(numbers, temp, left, mid+1, right);

}

}

void merge(int numbers[], int temp[], int left, int mid, int right)

{

inti, left\_end, num\_elements, tmp\_pos; left\_end = mid - 1;

tmp\_pos = left;

num\_elements = right - left + 1;

while ((left <= left\_end) && (mid <= right))

{

if (numbers[left] <= numbers[mid])

{

temp[tmp\_pos] = numbers[left]; tmp\_pos = tmp\_pos + 1;

}

else

{

}

}

left = left +1;

temp[tmp\_pos] = numbers[mid]; tmp\_pos = tmp\_pos + 1;

mid = mid + 1;

while (left <= left\_end)

{

temp[tmp\_pos] = numbers[left]; left = left + 1;

tmp\_pos = tmp\_pos + 1;

}

while (mid <= right)

{

temp[tmp\_pos] = numbers[mid]; mid = mid + 1;

tmp\_pos = tmp\_pos + 1;

}

for (i=0; i<= num\_elements; i++)

{

numbers[right] = temp[right]; right = right - 1;

}

}

Input: 13 -5

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| -8 | 15 | 60 | 17 | 31 47 |  |
| -5 | 13 | 15 | 17 | 31 47 | 60 |

Output:

Sorted numbers are: -8

Conclusion:

Performance Analysis of Merge Sort

Let T(n) be the time taken by this algorithm to sort an array of n elements dividing A intosub arrays A1 and A2 takes linear time. It is easy to see that the Merge (A1, A2, A) also takes the linear time. Consequently,

T(n) = T(n/2) + T(n/2) +θ(n)

T(n) = 2T (n/2) + θ(n)

The total running time of Merge sort algorithm is O(n lgn), which is asymptotically optimal like Heap sort, Merge sort has a guaranteed n lgn running time. Merge sort required (n) extra space. Merge is not in-place algorithm .The only known ways to merge in-place (without any extra space) are too complex to be reduced to practical program.