**EXPERIMENT 1 : DIVIDE AND CONQUER**

**AIM:** Write a c program to implement the following algorithms using divide and conquer method

1. Merge Sort and Binary Search
2. MinMax and Quick Sort
3. Find the kth smallest element
4. Strassen’s Matrix Multiplication

**Theory:**

The Divide-and-Conquer strategy suggests splitting the inputs into k distinct subsets, 1< k≤n, yielding k subproblems. These subproblems must be solved, and then a method must be found to combine sub-solutions into a solution of the whole.

If the sub problems are still relatively large, then the divide-and-conquer strategy can possibly be reapplied. DAndC is initially invoked as DAndC(P), where P is the problem to be solved.

Small(P)is a Boolean-valued function that determines whether the input size is small enough that the answer can be computed without splitting. The subproblems P₁, P2..... Pk Combine is a function that determines the solution to P using the solutions to the k subproblem.

For divide-and-conquer based algorithms that produce sub-problems of the same type as the original problem, it is very natural to first describe such algorithms using recursion.

Some standard algorithms that follow Divide and Conquer algorithm are:-

Quick Sort, Merge Sort, Find MinMax elements, Strassen’s Matrix Multiplication etc.

**ALGORITHMS:**

A] Merge Sort:

Algorithm MergeSort(low, high)

   // a[low:high] is a global array to be sorted. small(P) is true if there is only 1 element to sort.

  //In this case the list is already sorted.

{

if(low < high) then. // If there are more than 1 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 solutions.

Merge(low, mid, high);

}

}

Algorithm Merge(low, mid, high)

  // a[low:high] is a global array containing 2 sorted subsets in a[low:mid] and in a[mid:high].

  //The goal is to merge these 2 sets into a single set residing in a[low:high]. b[] is an

  // auxiliary set.

{

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];0}

B] Binary Search:

**Iterative Binary Search Algorithm**

Algorithm BinarySearch(a, n, x)

// Given an array a[1:n] of elements in ascending order, n>=0

//determine whether x is present, and if so, return j such that x=a[j]; else return 0.

{

low:=1; high:=n;

while(low <= high) do

{

mid:=[(low+high)/2];

if(x < a[mid]) then high:=mid-1;

else if(x > a[mid]) then low:=mid+1;

else return mid;

}

return 0;

}

**Recursive Binary Search Algorithm**

Algorithm BinarySearch(a, I, l, x)

// Given an array a[i:l] of elements in ascending order, 1<= I <= l,

//determine whether x is present, and if so, return j such that x=a[j]; else return 0.

{

if(l = i) then //If Small(P)

{

if(x = a[I]) then return I;

else return 0;

}

else

{

// Reduce P into a smaller subproblem.

mid:=[(I+l)/2];

if(x = a[mid]) then return mid;

else if (x < a[mid]) then

return BinarySearch(a, i, mid-1, x);

else return BinarySearch(a, mid+1, l, x);

}

}

C] Min-Max Algorithm:

Algorithm Min-max(i, j, max, min)

// a[1:n] is a global array

// Parameter i, j are integers

// 1<=i<=j<=n

// max stores the largest element and min stores the smallest element of a[1:n]

{

if(i==j) then max:=min:=a[i];

else if (i=j-1) then

{

if(a[i]>a[j]) then

{

max:=a[i], min:=a[j];

}

else

{

max:=a[j], min:=a[i];

}

}

else

{

mid=⌊(i+j)/2⌋;

Min-Max(i, mid, max, min);

Min-Max(mid+1, j, max1, min1);

if(max1>max) then max:=max1;

if(min1<min) then min:=min1;

}

}

D] Quick Sort Algorithm:

Algorithm QuickSort(arr[], i, j)

// i -> starting index

// j -> upper index

// arr[] -> array to be sorted

{

if(i<j) then

{

p:=partition(arr, i, j);

QuickSort(arr, i, p-1);

QuickSort(arr, p+1, j);

}

}

Algorithm partition(arr, l, h)

{

Pivot:=arr[h];

i:=l;

for j:=low to h-1 do

{

if(arr[j]<pivot) then

{

temp:=arr[j];

arr[j]:=arr[i];

arr[i]:=temp;

i:=i+1;

}

}

temp:=arr[j];

arr[j]:=arr[i];

arr[i]:=temp;

return i;

}

E] Kth smallest elment:

Algorithm ksmall(a[], l, r, k)

// selects the kth smallest element in a[1:n] and places it in the kth position

{

if (r<k) then return;

repeat

{

p=partition(a[], l, r);

if (p=k) then return;

else

{

if(p>k) then

r:=p-1;

else

l:=p+1;

}

} until (l<=r);

}

Algorithm partition(arr, l, h)

{

Pivot:=arr[h];

i:=l;

for j:=low to h-1 do

{

if(arr[j]<pivot) then

{

temp:=arr[j];

arr[j]:=arr[i];

arr[i]:=temp;

i:=i+1;

}

}

temp:=arr[j];

arr[j]:=arr[i];

arr[i]:=temp;

return i;

}

**Time and Space Complexity:**

● 1a]

● The time complexity of the binary search algorithm is O(log n).

● The best-case time complexity would be O(1) when the central index would directly match

the desired value.

● The worst-case scenario could be the values at either extremity of the list or values not in the

list.

● The time complexity of Binary Search can be written as

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

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 : o According to master’s theorem

a=1,b=2 n^(log(base

b)a)=n^(0) =1 T(n) =

n^(0)\*U(n)

U(n)->h(n)=F(n)/(n^0)

= c/(n^0) =>c=>r=0

=>i=0

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

Therefore, the time complexity is T(n) = 1\*Log(n)=O(Log(n))

1b]

Merge Sort is a recursive algorithm. If the time for merging operation is proportional to n, then the

computing time for merge sort is described by the recurrence relation:

T(n) = 2T(n/2) + cn, n>1, c is a constant.

T(1) = a, where ‘a’ is a constant

Applying repeated substitution method:

When n is a power of 2, n = 2k,

=> log n = log2k

=> log2n = k

So we can replace k with log2n

We can solve the equation as follows:

T(n) = 2(2T(n/4) + cn/2) + cn

= 4T(n/4) + 2cn

= 4(2T(n/8) + cn/4) + 2cn

.

.

= 2k T(1) + kcn

= an + cn\*log2 n

We see that 2k < n < 2k+1, then T(n) <= T(2k+1)

Therefore, T(n) = Θ(n log n)

Space Complexity: O(n)

2a] Let T(n) represent the number of comparisons needed for MaxMin, then the recurrence relation is:

T(n) = 0 if n=1

=1 if n=2

=T(n/2)+T(n/2)+2 if n>2

T(n)=2T(n/2)+2 --- (1)

From (1)

T(n/2)=2T(n/4)+2 -----(2)

Substitute (2) in (1)

T(n)=2[2T(n/4)+2]+2 =4T(n/4)+4+2----(3)

From (1)

T(n/4)=2T(n/8)+2 ------(4)

Substitute (4) in (2)

T(n)= 4[2T(n/48+2]+4+2

=8t(n/8)+8+4+2

.....

.....

......

2^k-1 T(2^k/2^k-1)+2^k-2

2^k-1 T(2)+2^k-2

=2^k-1+2^k-2

=n/2+n-2

=3n/2-2

=O(n)

Space Complexity: O(1)

2b]

Let T(n) represent the number of comparisons needed for Quicksort, then the recurrence relation is:

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

T(n) = 1 if n=1

= 2 T(n/2)+n if n>1

T(n)=2T(n/2)+n --- (1)

T(n/2)=2T(n/4)+(n/2) -----(2)

Substitute (2) in (1)

T(n)=2[2T(n/4)+/(n2)]+n =4T(n/4)+2n----(3)

By substituting T(n/4) in ---(3)

T(n)=8T(n/8)+3n

T(n)= nT(n/n)+ logn\*n

T(n) =n+nlogn= O(nlogn)

This represents best and avg case.

The worst case is O(n^2).

Space Complexity: O(logn)

3a] Time Complexity: The worst-case time complexity for this algorithm is O(n), but it can

be improved if we choose the pivot element randomly. If we randomly select the pivot, the

expected time complexity would be linear, O(n).

Space Complexity: O(logn) average for recursion call stack

3b]

Time Complexity:

This algorithm takes O(nm) time where n is the length of the string and m is the length of the

substring to be searched. n time to loop through the characters of the string and m time is taken to

calculate the previous index of the mismatched character in the substring.

Space Complexity:

Takes auxiliary space of O(1) or total space complexity of O(n+m) where n, m are the lengths

of the string, substring respectively if we consider the input sizes.