**EX - 6: Divide and Conquer Approach**

**Aim:** Implementation of different real time problems using Divide and Conquer approach.

1. Max-min problem
2. Merge sort
3. Quick sort

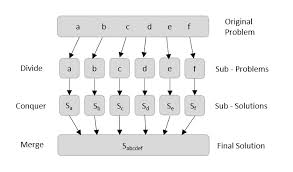
**Program Description:**

Divide and conquer is a problem-solving technique used to solve problems by dividing the main problem into subproblems, solving them individually and then merging them to find solution to the original problem.

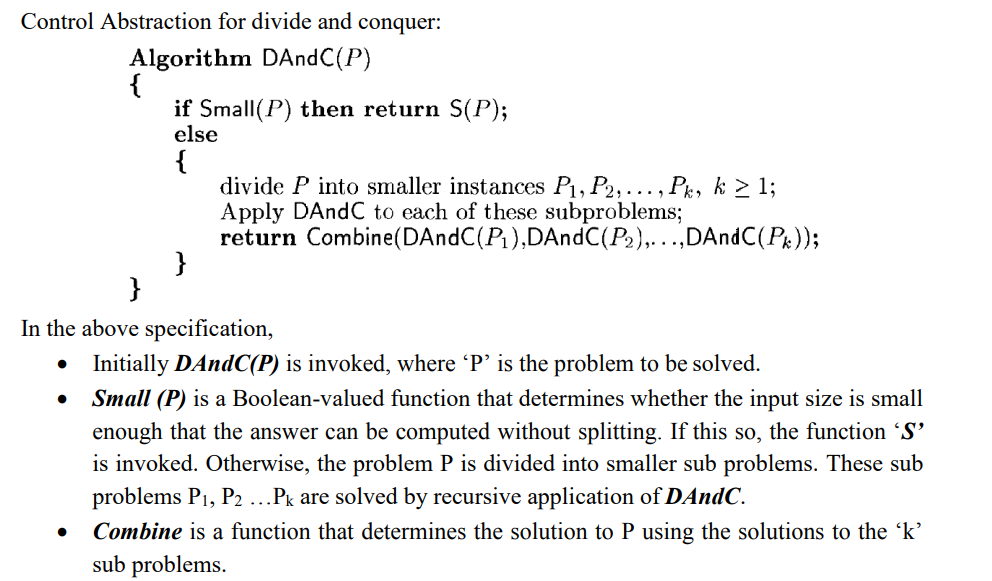
**Working of Divide and Conquer Algorithm:**

Divide and Conquer Algorithm can be divided into three steps:

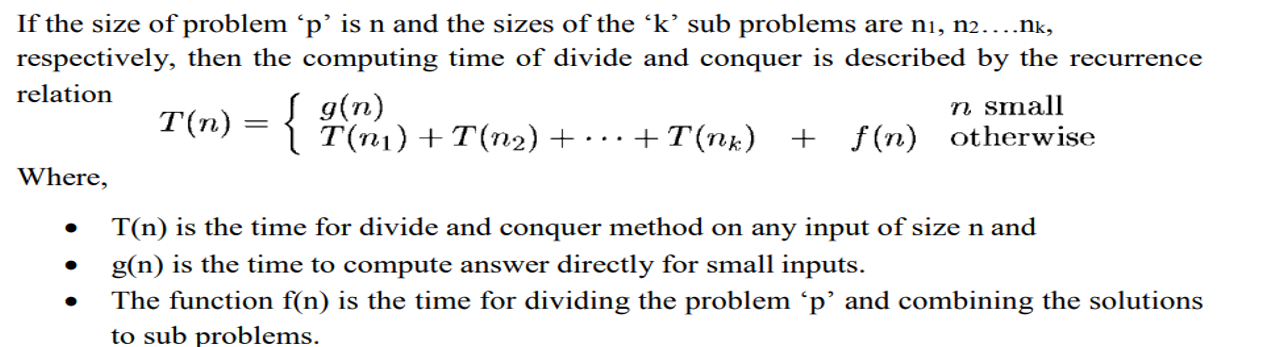
1. Divide,:
   * Break down the original problem into smaller subproblems.
   * Each subproblem should represent a part of the overall problem.
   * The goal is to divide the problem until no further division is possible.
2. Conquer :
   * Solve each of the smaller subproblems individually.
   * If a subproblem is small enough (often referred to as the “base case”), we solve it directly without further recursion.
   * The goal is to find solutions for these subproblems independently.
3. Merge:
   * Combine the sub-problems to get the final solution of the whole problem.
   * Once the smaller subproblems are solved, we recursively combine their solutions to get the solution of larger problem.
   * The goal is to formulate a solution for the original problem by merging the results from the subproblems.



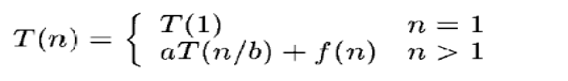
**General Algorithm specification of Divide and Conquer approach:**

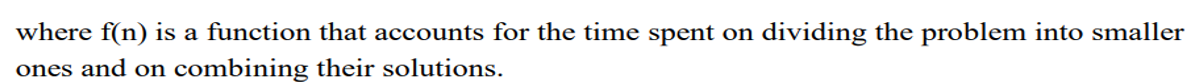
****

**Recurrence relation for Divide and Conquer approach:**

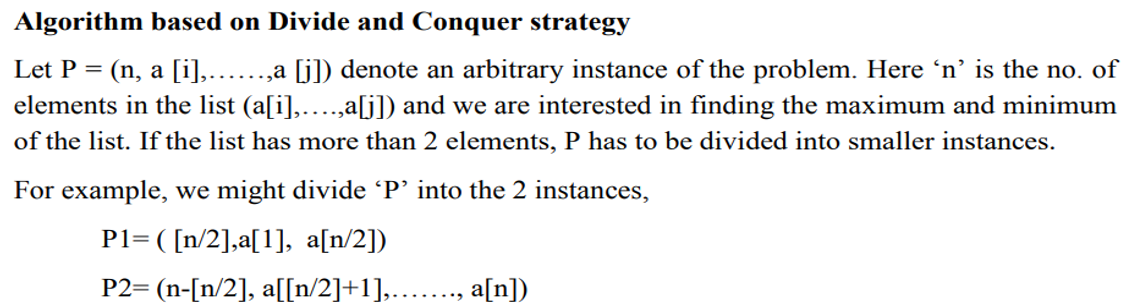
****

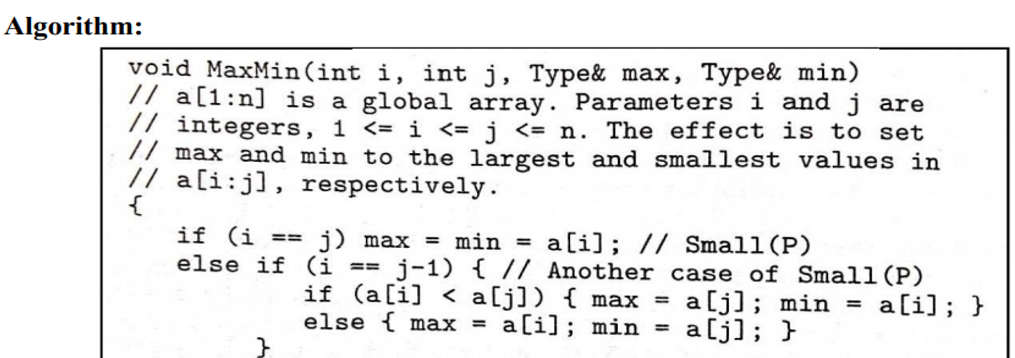
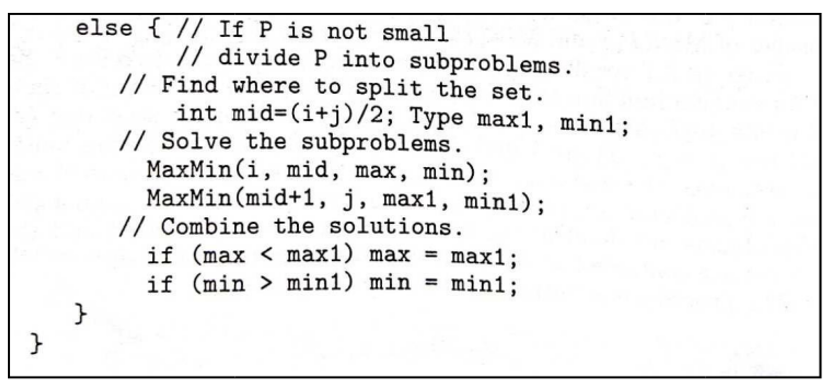
**In general it could represented as**

****

****

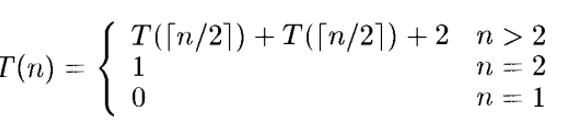
**Applications of Divide and Conquer approach;**

1. **Implementation of Max-Min problem:**

****

Program Code:

Input and Output:

**Recurrence relation for Max-Min problem:**

When solving this using substitution approach, the best, average and worst number of comparisons when n is power of two is 3n/2-2 instead of 2(n-1). Compared to the straight forward approach, this method saves 25% comparisons.

1. **Merge Sort:**

Merge sort uses the concept of sorting by merging. Merge-sort is based on the divide-and-conquer paradigm. It involves the following three steps:

1. Divide the array into two (or more) subarrays using recurssion
2. Sort each subarray
3. Merge them into one.

**Algorithm:**

procedure MERGE\_SORT(a, first, last)

/\*a[first:last] is the unsorted list of elements to be merge sorted. The call to the procedure to sort the list a[1:n] would be MERGE\_SORT(a,1,n) \*/

if (first < last) then

{

mid=(first + last)/2; /\* divide the list into two sublists\*/

MERGE\_SORT(a, first, mid); /\* merge sort the sublist a[first,mid]\*/

MERGE\_SORT(a, mid+1,last); /\* merge sort the sublist a[mid+1,last]\*/

MERGE(a, first, mid, last); /\* merge the two sublists a[first,mid] and a[mid+1,last] \*/

end MERGE\_SORT.

procedure MERGE (x, first, mid, last)

/\* x[first:mid] and x[mid+1:last] are ordered lists of data elements to be merged into a single ordered list x[first:last] \*/

first1=first;

last1=mid;

first2=mid + 1;

last2=last; /\* set the beginning and the ending indexes of the two lists into the appropriate variables\*/

i = first; /\* i is the index variable for the temporary output list temp\*/

/\* begin pair wise comparisons of elements from the two lists\*/

while (first1<=last1) and (first2<=last2) do

case

: x [first1] < x[first2] : { temp[i]= x[first1];

first1=first1 + 1;

i = i + 1;

}

: x [first1] > x[first2] : { temp[i]= x[first2];

first2 = first2 + 1;

i = i + 1;

}

: x [first1] = x[first2] : { temp[i]= x[first1];

temp[i + 1] = x[first2];

first1 = first1 + 1;

first2 = first2 + 1;

i = i + 2;

}

end /\*end case\*/

end /\* end while\*/

/\* the first list gets exhausted\*/

while(first2 <= last2) do

temp[i]= x[first2];

first2 = first2 + 1;

i = i + 1;

end

/\* the second list gets exhausted\*/

while (first1<= last1) do

temp[i]= x[first1];

first1 = first1 + 1;

i = i + 1;

end

/\* copy list temp to list x\*/

for j = first to last do

x[j] = templ[j];

end

end MERGE.

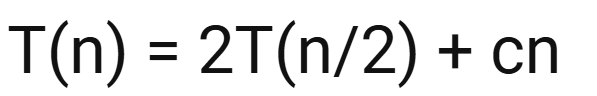
Program code:

Input and Output:

**Recurrence relation for Merge sort: (best, average and worst cases)**

**N=0 , T(1)**

**N=1, T(1)**

**When N>2,**

1. **. Quick Sort:**

Quick sort works on the principle sorting by exchange or transposition. Quick sort is a divide and conquer algorithm. Quick sort first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quick sort can then recursively sort the sub-arrays. The steps are:

1. Pick an element, called a pivot, from the array.
2. Reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
3. Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

**Algorithm:**

Procedure QUICK\_SORT(L, first, last )

/\* L[first:last] is the unordered list of elements to be quick sorted. The call to the procedure to sort the list L[1:n] would be QUICK\_SORT(L, 1, n)\*/

if (first < last) then

{ PARTITION(L, first, last, loc) ; /\* partition the list into two sublists at

loc\*/

QUICK\_SORT(L, first, loc-1 ); /\* quick sort the sublist L[first,loc-1]\*/

QUICK\_SORT(L, loc+1, last ): /\* quick sort the sublist L[loc+1, last]\*/

}

end OUICK\_SORT.

procedure PARTITION (L, first, last, loc )

/\* L[first:last] is the list to be partitioned. loc is the position where the pivot element finally settles down\*/

left = first;

right = last+1;

pivot\_elt = L[first]; /\* set the pivot element to the first element in list L\*/

while (left < right) do

repeat

left = left+1; /\* pivot element moves left to right\*/

until L[left] >= pivot\_elt;

repeat

right = right -1; /\* pivot element moves right to left\*/

until L[right] <= pivot\_elt;

if (left < right) then

swap(L[left], L[right]); /\*arrows face each other\*/

end

loc = right

swap(L[first], L[right]); /\* arrows have crossed each other – exchange pivot element L[first] with L[right] \*/

end PARTITION.

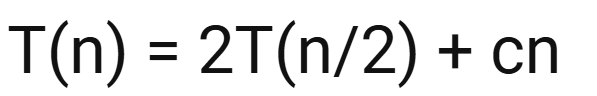
**Program code:**

**Input and Output:**

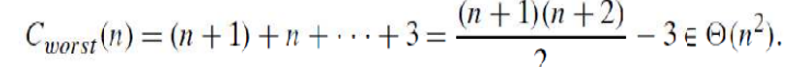
**Recurrence relation for Quicksort sort: (Best, average cases)**

**N=0 , T(1)**

**N=1, T(1)**

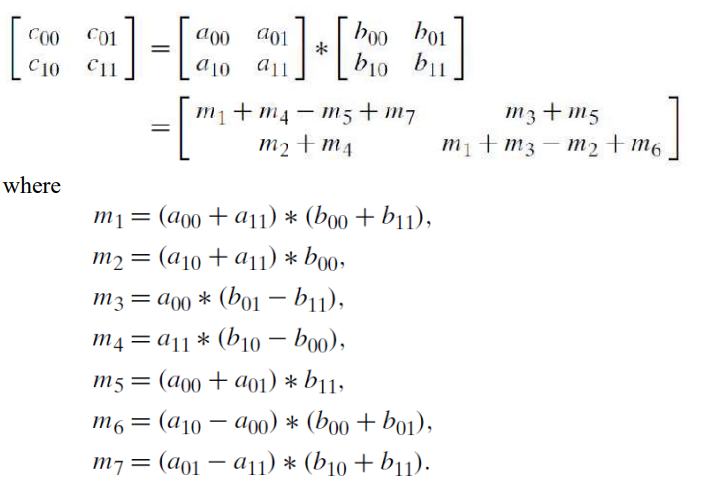
**When N>2,**

Worst case:



1. **Implementation of Strassen’s Matrix multiplication:**

By using divide-and-conquer approach proposed by Strassen in 1969, we can reduce the number of multiplications. Multiplication of 2×2 matrices: The principal insight of the algorithm lies in the discovery that we can find the product C of two 2 × 2 matrices A and B with just 7 multiplications as opposed to the eight required by the brute-force algorithm. This is accomplished by using the following formulas:



Thus, to multiply two 2×2 matrices, Strassen’s algorithm makes seven multiplications and 18 additions/subtractions, whereas the brute-force algorithm requires eight multiplications and four additions.

**Program code:**

**Input and Output:**

**Time complexity analysis:**

