UNIT - III
DIVIDE AND CONQUER

General method:

- Given a function to compute on 'n' inputs the divide-and-conquer strategy suggests splitting the inputs into 'k' distinct subsets, $1<k<=n$, yielding 'k' sub problems.

- These sub problems 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.

- Often the sub problems resulting from a divide-and-conquer design are of the same type as the original problem.

- For those cases the re application of the divide-and-conquer principle is naturally expressed by a recursive algorithm.

D And C(Algorithm) is initially invoked as D and C(P), where 'p' is the problem to be solved. Small(P) is a Boolean-valued function that determines whether the i/p size is small enough that the answer can be computed without splitting. If this so, the function 'S' is invoked. Otherwise, the problem P is divided into smaller sub problems. These sub problems P1, P2 ...Pk are solved by recursive application of D And C. Combine is a function that determines the solution to p using the solutions to the 'k' sub problems.

- If the size of 'p' is n and the sizes of the 'k' sub problems are n1, n2 ....n_k, respectively, then the computing time of D And C is described by the recurrence relation.

$$T(n)= \begin{cases} g(n); & \text{n small} \\ T(n1)+T(n2)+...............+T(nk)+f(n); & \text{otherwise.} \end{cases}$$

Where T(n) → is the time for D And C on any I/p of size 'n'.
g(n) → is the time of compute the answer directly for small I/ps.
f(n) → is the time for dividing P & combining the solution to sub problems.

```
Algorithm D And C(P)
{
     if small(P) then return S(P);
     else
     {
          divide P into smaller instances P1, P2... Pk, k>=1;
          Apply D And C to each of these sub problems;
        return combine (D And C(P1), D And C(P2),,D And C(Pk));
      }
}
```

- The complexity of many divide-and-conquer algorithms is given by recurrences
  of the form

$$T(n) = \begin{cases} T(1) & n=1 \\ aT(n/b)+f(n) & n>1 \end{cases}$$

→ Where a & b are known constants.
→ We assume that T(1) is known & 'n' is a power of b(i.e., n=b^k)

- One of the methods for solving any such recurrence relation is called the substitution method.
- This method repeatedly makes substitution for each occurrence of the function. T is the Right-hand side until all such occurrences disappear.

Example:

1) Consider the case in which a=2 and b=2. Let T(1)=2 & f(n)=n.
   We have,

$$\begin{aligned} T(n) &= 2T(n/2)+n \\ &= 2[2T(n/2/2)+n/2]+n = [4T(n/4)+n]+n \\ &= 4T(n/4)+2n \\ &=4[2T(n/4/2)+n/4]+2n = 4[2T(n/8)+n/4]+2n \\ &= 8T(n/8)+n+2n \\ &= 8T(n/8)+3n \\ &\quad * \\ &\quad * \end{aligned}$$

In general, we see that $T(n)=2^i\,T(n/2^i)+in.$, for any log n >=I>=1.

→ $T(n) = 2^{\log n}\,T(n/2^{\log n}) + n \log n$

→Corresponding to the choice of i=log n

→ Thus, $T(n) = 2^{\log n}\,T(n/2^{\log n}) + n \log n$

$$\begin{aligned} &\ldots \\ &= n.\ T(n/n) + n \log n \\ &= n.\ T(1) + n \log n \qquad \text{[since, log 1=0, 2^0=1]} \\ &= 2n + n \log n \end{aligned}$$

BINARY SEARCH

Binary search is a problem of determining whether a given element is present in the list of elements that are sorted in ascending order.
Let a_i, $1 \le i \le n$, be the list of elements that are sorted in ascending order. If the given element x is present in a list, we are to determine a value j such that a_j=x. If x is not in the list, the j is set to be zero.

Algorithm for Recursive Binary Search:

Algorithm BinSrch (*a, i, l, x*)
//Given an array *a*[*i : l*] of elements in nondecreasing
//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 sub problem.
      mid := └(i+l)/2┘ ;
    if (x = a[mid]) then return mid;
    else if (x < a[mid]) then
            return BinSrch (a, i, mid -1, x);
    else return BinSrch(a, mid+1, i, x);
    }
}
```

<u>Algorithm for Iterative Binary Search</u>:

```
Algorithm Binsearch(a,n,x)
// Given an array a[1:n] of elements in non-decreasing
//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;
     else if(x>a[mid]) then
          low=mid+1;
  else return mid;
 }
 return 0;
}
```

Algorithm Binsrch describes this binary search method, where Binsrch has 4 inputs a[], i , l & x. It is initially invoked as Binsrch (a,1,n,x)

A non-recursive version of Binary search algorithm Binsearch has 3 inputs a,n,x. The while loop continues processing as long as there are more elements left to check. At the conclusion of the procedure 0 is returned if x is not present, or 'j' is returned, such that a[j]=x. We observe that low & high are integer Variables such that each time through the loop either x is found or low is increased by at least one or high is decreased at least one. Thus we have 2 sequences of integers approaching each other and eventually low becomes > than high & causes termination in a finite no. of steps if 'x' is not present.

Example:

1) Let us select the 14 entries.
   -15,-6,0,7,9,23,54,82,101,112,125,131,142,151.

→ Place them in a[1:14], and simulate the steps Binsearch goes through as it searches for different values of 'x'.

→ Only the variables, low, high & mid need to be traced as we simulate the algorithm.

→ We try the following values for x: 151, -14 and 9.
for 2 successful searches & 1 unsuccessful search.

- Table. Shows the traces of Bin search on these 3 steps.

X=151	low	high	mid
	1	14	7
	8	14	11
	12	14	13
	14	14	14
			Found

x=-14	low	high	mid
	1	14	7
	1	6	3
	1	2	1
	2	2	2
	2	1	Not found

x=9	low	high	mid
	1	14	7
	1	6	3
	4	6	5
			Found

Theorem: Algorithm Binsearch(a,n,x) works correctly.

Proof:
We assume that all statements work as expected and that comparisons such as x>a[mid] are appropriately carried out.

- Initially low =1, high= n,n>=0, and a[1]<=a[2]<=........<=a[n].
- If n=0, the while loop is not entered and is returned.
- Otherwise we observe that each time thro' the loop the possible elements to be checked of or equality with x and a[low], a[low+1],........,a[mid],......a[high].
- If x=a[mid], then the algorithm terminates successfully.
- Otherwise, the range is narrowed by either increasing low to (mid+1) or decreasing high to (mid-1).
- Clearly, this narrowing of the range does not affect the outcome of the search.
- If low becomes > than high, then 'x' is not present & hence the loop is exited.

MERGE SORT

Merge sort is an example of divide-and-conquer, it is a sorting algorithm that has the nice property that is in the worst case its complexity is O(n log n)

- This algorithm is called merge sort
- We assume that the elements are to be sorted in non-decreasing order.
- Given a sequence of 'n' elements a[1],...,a[n] the general idea is to imagine then split into 2 sets a[1],.....,a[n/2] and a[[n/2]+1],....a[n].

- Each set is individually sorted, and the resulting sorted sequences are merged to produce a single sorted sequence of ‘n’ elements.
- Thus, we have another ideal example of the divide-and-conquer strategy in which the splitting is done into 2 equal-sized sets & the combining operation is the merging of 2 sorted sets into one.

Algorithm For 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 than 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 solutions .
merge(low,mid,high);
}
}
```

Algorithm: Merging 2 sorted subarrays using auxiliary storage.

```
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 2 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 = 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];
}
```

- Consider the array of 10 elements a[1:10] =(310, 285, 179, 652, 351, 423, 861, 254, 450, 520)

- Algorithm Merge sort begins by splitting a[] into 2 sub arrays each of size five (a[1:5] and a[6:10]).
- The elements in a[1:5] are then split into 2 sub arrays of size 3 (a[1:3]) and 2(a[4:5])
- Then the items in a a[1:3] are split into sub arrays of size 2 a[1:2] & one(a[3:3])
- The 2 values in a[1:2} are split to find time into one-element sub arrays, and now the merging begins.

(310| 285| 179| 652, 351| 423, 861, 254, 450, 520)

→ Where vertical bars indicate the boundaries of sub arrays.

→Elements a[I] and a[2] are merged to yield,
(285, 310|179|652, 351| 423, 861, 254, 450, 520)

→ Then a[3] is merged with a[1:2] and
(179, 285, 310| 652, 351| 423, 861, 254, 450, 520)

→ Next, elements a[4] & a[5] are merged.
(179, 285, 310| 351, 652 | 423, 861, 254, 450, 520)

→ And then a[1:3] & a[4:5]
(179, 285, 310, 351, 652| 423, 861, 254, 450, 520)

→ Repeated recursive calls are invoked producing the following sub arrays.
(179, 285, 310, 351, 652| 423| 861| 254| 450, 520)

→ Elements a[6] &a[7] are merged.

→Then a[8] is merged with a[6:7]
(179, 285, 310, 351, 652| 254,423, 861| 450, 520)

→ Next a[9] &a[10] are merged, and then a[6:8] & a[9:10]
(179, 285, 310, 351, 652| 254, 423, 450, 520, 861)

→ At this point there are 2 sorted sub arrays & the final merge produces the fully sorted result.

(179, 254, 285, 310, 351, 423, 450, 520, 652, 861)

- If the time for the merging operations is proportional to 'n', then the computing time for merge sort is described by the recurrence relation.

$$T(n) = \begin{cases} a & n=1, \quad \text{'a' a constant} \\ 2T(n/2)+cn & n>1, \quad \text{'c' a constant.} \end{cases}$$

→ When 'n' is a power of 2, $n= 2^k$, we can solve this equation by successive substitution.

$$\begin{aligned} T(n) &= 2(2T(n/4) + cn/2) + cn \\ &= 4T(n/4)+2cn \\ &= 4(2T(n/8)+cn/4)+2cn \\ &\quad * \\ &\quad * \\ &= 2^k\, T(1)+kCn. \\ &= an + cn \log n. \end{aligned}$$

→ It is easy to see that if $s^k<n<=2^k+1$, then $T(n)<=T(2^k+1)$. Therefore,

T(n)=O(n log n)

QUICK SORT

- The divide-and-conquer approach can be used to arrive at an efficient sorting method different from merge sort.

- In merge sort, the file a[1:n] was divided at its midpoint into sub arrays which were independently sorted & later merged.

- In Quick sort, the division into 2 sub arrays is made so that the sorted sub arrays do not need to be merged later.

- This is accomplished by rearranging the elements in a[1:n] such that a[I]<=a[j] for all I between 1 & n and all j between (m+1) & n for some m, 1<=m<=n.

- Thus the elements in a[1:m] & a[m+1:n] can be independently sorted.

- No merge is needed. This rearranging is referred to as partitioning.

- Function partition of Algorithm accomplishes an in-place partitioning of the elements of a[m:p-1]

- It is assumed that a[p]>=a[m] and that a[m] is the partitioning element. If m=1 & p-1=n, then a[n+1] must be defined and must be greater than or equal to all elements in a[1:n]

- The assumption that a[m] is the partition element is merely for convenience, other choices for the partitioning element than the first item in the set are better in practice.

- The function interchange (a,I,j) exchanges a[I] with a[j].

Algorithm: Partition the array a[m:p-1] about a[m]

```
Algorithm Partition(a,m,p)
//within a[m],a[m+1],.....,a[p-1] the elements
// are rearranged in such a manner that if
//initially t=a[m],then after completion
//a[q]=t for some q between m and
//p-1,a[k]<=t for m<=k<q, and
//a[k]>=t for  q<k<p. q is returned
//Set a[p]=infinite.
{
v=a[m];I=m;j=p;
repeat
{
   repeat
       I=I+1;
   until(a[I]>=v);
   repeat
       j=j-1;
   until(a[j]<=v);
   if (I<j) then Interchange(a,i.j);
}until(I>=j);
   a[m]=a[j]; a[j]=v;
   retun j;
}
```

```
Algorithm Interchange(a,I,j)
//Exchange a[I] with a[j]
{
   p=a[I];
   a[I]=a[j];
   a[j]=p;
}
```

Algorithm: Sorting by Partitioning

```
Algorithm Quicksort(p,q)
//Sort the elements a[p],....a[q] which resides
//is the global array a[1:n] into ascending
//order; a[n+1] is considered to be defined
// and must be >= all the elements in a[1:n]
{
if(p<q) then // If there are more than one element
{
// divide p into 2 subproblems
j= Partition(a,p,q+1);
```

11.//'j' is the position of the partitioning element.
12.//solve the subproblems.
13.Quicksort(p,j-1);
14.Quicksort(j+1,q);
15.//There is no need for combining solution.
16.}
17.}

Input: Unsorted list of elements
Output: Sorted list of elements
Example:
Consider the list

65 45 50 55 85 60 80 75 70 ∞
Pivot & I j

65 70 75 80 85 60 55 50 45 ∞
Pivot I j

Since i<J, swap a[i] and a[j] i.e 70 and 45,
65 45 75 80 85 60 55 50 70 ∞
Pivot I j

Since i<J, swap a[i] and a[j] i.e 75 and 50
65 45 50 80 85 60 55 75 70 ∞
Pivot I j

Since i<J, swap a[i] and a[j] i.e 80 and 55
65 45 50 55 85 60 80 75 70 ∞
Pivot i j

Since i<J, swap a[i] and a[j] i.e 85 and 60
65 45 50 55 60 85 80 75 70 ∞
Pivot j I

Since i>j swap a[j] with pivot element i.e., 60 and 65 and now partition occurs
60 45 50 55 65 85 80 75 70 ∞

List is divided into three sublists:
List1: 60 45 50 55 (Elements less than pivot)
List2: 65 (Elements equal to pivot)
List3: 85 80 75 70 (Elements greater than pivot)
QuickSort is again applied for List1 and List2.
Average Time Complexity of Quick Sort:

Let the average case value be $T_A(n)$.
Under the assumptions, the partitioning element v has an equal probability of being the ith smallest element, $1 \le i \le p-m$ in a[m:p-1]. Hence the two subarrays remaining to be sorted are a[m:j] and a[j+1:p-1] with probability 1/(p-m), $m \le j < p$.

From this recurrence obtained is

$$T_A(n) = n+1+1/n \sum_{1\le k\le n}[T_A(k-1) + T_A(n-k)] \qquad \text{--------} \quad 1$$

The no. of element comparisons required by Partition algorithm on its first call is n+1.

Note $T_A(0)=T_A(1)=0$ -------- 2

Multiplying both sides of 1 by n, we get,

$$nT_A(n) = n(n+1)+ \sum_{1\le k\le n}[T_A(k-1) + T_A(n-k)]$$

$$\Rightarrow nT_A(n)= n(n+1)+2[TA(0)+TA(1)+ \dots +TA(n-1)] \qquad \text{--------} \quad 3$$

Repalacing n by n-1 in 3, we get,

$$\Rightarrow (n-1)T_A(n-1)= n(n-1)+2[TA(0)+TA(1)+ \dots +TA(n-2)] \qquad \text{--------} \; 4$$

Substracting 3 -4, we get,

$nT_A(n) - (n-1)T_A(n-1) = 2n + 2T_A(n-1)$

$\Rightarrow T_A(n)/(n+1) = T_A(n-1)/n + 2/(n+1)$

By substitution method,

$$\begin{aligned} T_A(n)/(n+1) &= T_A(n-2)/n-1 + 2/n + 2/n+1 \\ &= T_A(n-3)/n-2 + 2/n-1 + 2/n + 2/n+1 \\ &\quad \vdots \\ &= T_A(1)/2 + 2 \textstyle\sum_{3\le k\le n+1} 1/k \\ &= 2 \textstyle\sum_{3\le k\le n+1} 1/k \end{aligned}$$

Since,

$$\sum_{3\le k\le n+1} 1/k \le \int_2^{n+1} 1/x \, dx = \log_e (n+1) - \log_e 2$$

Therefore, $T_A(n) \le 2(n+1)[\log_e (n+1) - \log_e 2] = O(n \log n)$.

STRASSEN'S MATRIX MULTIPLICAION

- Let A and B be the two n*n Matrix. The product matrix C=AB is calculated by using the formula,

  C (i ,j)= A(i,k) B(k,j) for all 'i' and and j between 1 and n.

- The time complexity for the matrix Multiplication is $O(n^3)$.

- Divide and conquer method suggest another way to compute the product of n*n matrix.

- We assume that N is a power of 2 .In the case N is not a power of 2, then enough rows and columns of zero can be added to both A and B. So that the resulting dimension are the powers of two.

- If n=2 then the following formula as a computed using a matrix multiplication operation for the elements of A & B.

- If n>2,Then the elements are partitioned into sub matrix n/2*n/2..since 'n' is a power of 2 these product can be recursively computed using the same formula .This Algorithm will continue applying itself to smaller sub matrix until 'N" become suitable small(n=2) so that the product is computed directly .
- The formula are

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{21} \end{pmatrix} * \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

$C_{11} = A_{11} * B_{11} + A_{12} * B_{21}$

$C_{12} = A_{11} * B_{12} + A_{12} * B_{22}$

$C_{21} = A_{21} * B_{11} + A_{22} * B_{21}$

$C_{22} = A_{21} * B_{12} + A_{22} * B_{22}$

For EX:

$$4 * 4 = \begin{pmatrix} 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 \end{pmatrix} * \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

The Divide and conquer method

$$\begin{pmatrix} \begin{vmatrix} 2 & 2 \\ 2 & 2 \end{vmatrix} & \begin{vmatrix} 2 & 2 \\ 2 & 2 \end{vmatrix} \\ \begin{vmatrix} 2 & 2 \\ 2 & 2 \end{vmatrix} & \begin{vmatrix} 2 & 2 \\ 2 & 2 \end{vmatrix} \end{pmatrix} \begin{pmatrix} \begin{vmatrix} 1 & 1 \\ 1 & 1 \end{vmatrix} & \begin{vmatrix} 1 & 1 \\ 1 & 1 \end{vmatrix} \\ \begin{vmatrix} 1 & 1 \\ 1 & 1 \end{vmatrix} & \begin{vmatrix} 1 & 1 \\ 1 & 1 \end{vmatrix} \end{pmatrix} = \begin{pmatrix} \begin{vmatrix} 4 & 4 \\ 4 & 4 \end{vmatrix} & \begin{vmatrix} 4 & 4 \\ 4 & 4 \end{vmatrix} \\ \begin{vmatrix} 4 & 4 \\ 4 & 4 \end{vmatrix} & \begin{vmatrix} 4 & 4 \\ 4 & 4 \end{vmatrix} \end{pmatrix}$$

- To compute AB using the equation we need to perform 8 multiplication of n/2*n/2 matrix and 4 addition of n/2*n/2 matrix.
- The time complexity for the above matrix Multiplication is $O(n^3)$.
- The overall computing time T(n) of the resulting divide and conquer algorithm is given by the sequence.

$$T(n) = \begin{cases} b & n \le 2 \quad a \,\&\, b \text{ are} \\ 8T(n/2) + cn^2 & n > 2 \quad \text{constant} \end{cases}$$

That is $T(n) = O(n^3)$

Matrix multiplication are more expensive then the matrix addition .We can attempt to reformulate the equation for C_{ij} so as to have fewer multiplication and possibly more addition .

- Strassen has discovered a way to compute the Cij using only 7 multiplication and 18 addition or subtraction.

Strassen's formula are

$P= (A_{11}+A_{12})(B_{11}+B_{22})$

$Q= (A_{12}+A_{22})B_{11}$

$R= A_{11}(B_{12}-B_{22})$

$S= A_{22}(B_{21}-B_{11})$

$T= (A_{11}+A_{12})B_{22}$

$U= (A_{21}-A_{11})(B_{11}+B_{12})$

$V= (A_{12}-A_{22})(B_{21}+B_{22})$

$C_{11}=P+S-T+V$

$C_{12}=R+t$

$C_{21}=Q+T$

$C_{22}=P+R-Q+V$

The resulting recurrence relation for T(n) is

$$T(n)= \begin{cases} b & n\le 2 \\ 7T(n/2)+an^2 & n>2 \end{cases} \quad \text{a \&b are constant}$$

By using substitution method,

$$T(n) = an^2 [1 + 7/4 + (7/4)^2 + \ldots + (7/4)^{k-1}] + 7^K T(1)$$

$$\le cn^2 (7/4) \log_2 n + 7^{\log 2\ n} \text{, c a constant}$$

$$= cn^{\log 2\ 4 + \log 2\ 7 - \log 2\ 4} + n \log_2 7$$

$$= O(n \log_2 7) \approx O(n^{2.81}).$$

* * * * *