Divide and Conquer Algorithms

General Method:

In divide and conquer method, a given problem is:

i) Divided into smaller sub-problems.

ii) These sub-problems are solved independently.

iii) Combining all the solutions of sub-problems into a solution of the whole.

→ If the sub-problems are large enough then divide and conquer is reapplied.

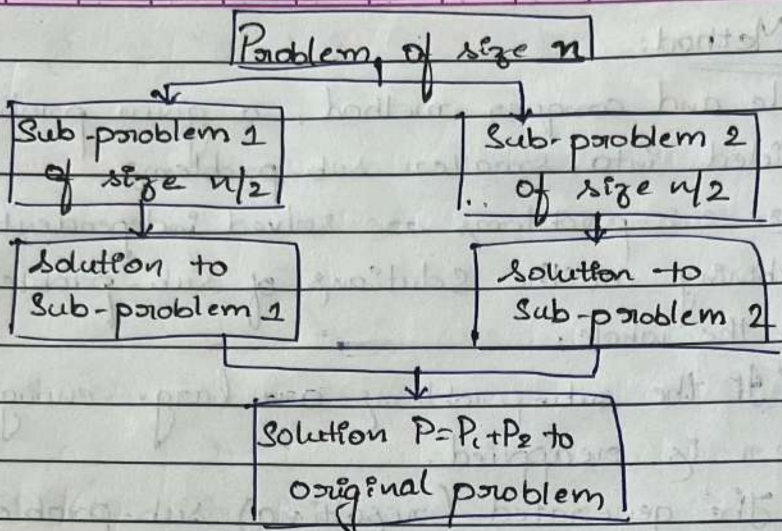
→ The generated (resulting) sub-problems from a divide and conquer design are of the same type as the original problem.

→ For those cases "recursive algorithms are used in divide & conquer strategy.

Control abstraction for divide and conquer:

* Using control abstraction a flow of control of a procedure is:

```
Algorithm Divide_Conquer(P)
{
  if P is too small then
  return solution to P
  else
  {
  Divide(P) and obtain P1, P2 ---Pn
  where n>1
  Apply DC to each sub-problem
     return Combine(DC(P1), DC(P2) -- DC(Pn));
  }
}
```



* Computing time of Divide-Conquer is given by the recurrence relation

$$T(n) = \begin{cases} g(n), & \text{if } n \text{ is small} \\ T(n_1)+T(n_2)+\dots+T(n_k)+f(n) & \text{Otherwise} \end{cases}$$

where $T(n)$ - time for Divide-Conquer on any input size n.

$g(n)$ - time to compute the answer directly for small inputs.

$f(n)$ - time required for dividing P and combining the solutions to sub-problems.

* For divide and Conquer based algorithms that produce sub-problems of the same type as the original problem, it is very natural to 1st describe such algorithms using recursion.

* 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 and b are known constants. We assume that $T(1)$ is known and n is a power of b (i.e., $n = b^k$)

* One of the methods for solving any such recurrence relation is called Substitution Method.

Eg: Consider the case in which $a=2$ and $b=2$.
Let $T(1) = 2$ and $f(n) = n$.

we have

$$T(n) = aT(n/b) + f(n)$$
$$= 2T(n/2) + n$$
$$= 2(2T(n/4) + n/2) + n$$
$$= 4T(n/4) + \cancel{2}(n/\cancel{2}) + n$$
$$= 4T(n/4) + 2n$$
$$= 4(2T(n/8) + n/4) + 2n$$
$$= 8T(n/8) + \cancel{4}(n/\cancel{4}) + 2n$$
$$= 8T(n/8) + 3n$$
$$= 2^3 T(n/2^3) + 3n$$
$$\vdots$$

upto k^{th} step $= 2^k T(n/2^k) + k \cdot n$

$= n \cdot T(n/n) + \log_2^n \cdot n$

$= n \cdot T(1) + n \log_2^n$

$= 2 \cdot n + n \log_2^n$

Assume $\therefore n = 2^k$ then $k = \log_2^n$

Hence, in terms of Big Oh (O) notation

$$\boxed{T(n) = O(n \log_2^n)}$$

Binary Search

* Binary Search is an efficient searching method.
* By using searching method, to search an element in a given list of elements.
* The list of elements that are sorted in "non-decreasing order".
* An element which is to be searched from the list of elements sorted in array A (i.e., a[0] --- a[n-1]) or a[1] --- a[n]) is called Key element.
* If key is searching element, it always choose the middle element of the array $mid = \frac{low+high}{2}$ then the resulting search algorithm is called as Binary Search.
* Let a[mid] be the middle element of array A. Then there are 3 conditions that needs to be tested while searching the array using this method:

   i) if key = a[mid] then desired element is present at middle of the list.

   ii) otherwise if key < a[mid] then search the element in left sublist of the middle element.

   iii) otherwise if key > a[mid] then search the element in right sublist of the middle element.

Representation

$$\underbrace{a[0] --- a[mid-1]}_{key < a[mid]} \quad \underbrace{a[mid]}_{key = a[mid]} \quad \underbrace{a[mid+1] --- a[n-1]}_{key > a[mid]}$$

Example.

0	1	2	3	4	5	6	7	8
10	12	24	29	39	40	51	56	69

$k = 56$

$$mid = \frac{beg + end}{2}$$

$beg = 0$; $end = 8$

$$mid = \frac{0+8}{2} = 4$$

0	1	2	3	4	5	6	7	8
10	12	24	29	39	40	51	56	59

↑

$a[mid] = 39$

$a[mid] < k$

$beg = mid + 1 = 5$, $end = 8$

$$mid = \frac{13}{2} = 6$$

10	12	24	29	39	40	51	56	69

↑

$a[mid] = 51$

$a[mid] < k$

$beg = mid + 1 = 7$, $end = 8$

$mid = 15/2 = 7$

10	12	24	29	39	40	51	56	69

↑

$a[mid] = 56$

$a[mid] = k$

Element found

So, algorithm will return the index of the element is matched

* 2 methods to implement → Iterative method
→ Recursive method

Algorithm (Iterative Binary Search)

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:= $\frac{(low+high)}{2}$;
if (x<a[mid]) then high:=mid-1;
else if (x>a[mid]) then low:=mid+1;
else return mid;
}
return 0;
}

Algorithm - Recursive Binary Search

Algorithm BinSearch (a, i, l, x)
// Given an array a[i:l] of elements in non-decreasing 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 (x=a[i]) then return i;
else return 0;
}

```
else
{
    mid := (i+l)/2 ;
    if (x= a[mid]) then return mid;
    else if (x<a[mid]) then
            return BinSearch(a, i, mid-1, x);
    else return BinSearch(a, mid+1, l, x);
  }
}
```

Analysis space:

Space Complexity: $S(P) = n+4$

Time Complexity: Best case : $\Theta(1)$

Avg case: $\Theta(\log n)$

Worst case: $\Theta(\log n)$

Algorithm for binary Search using one comparison per cycle:

```
Algorithm BinSearch (a, n, x)
{
 low:=1; high:= n+1;
 while( low < (high-1)) do
 {
  mid:= ⌊(low+high)/2⌋;
    if (x < a[mid]) then high:=mid;
  else low:=mid;
 }
 if(x=a[low]) then return low;
 else return 0;
}
```

TC: $\Theta(\log n)$

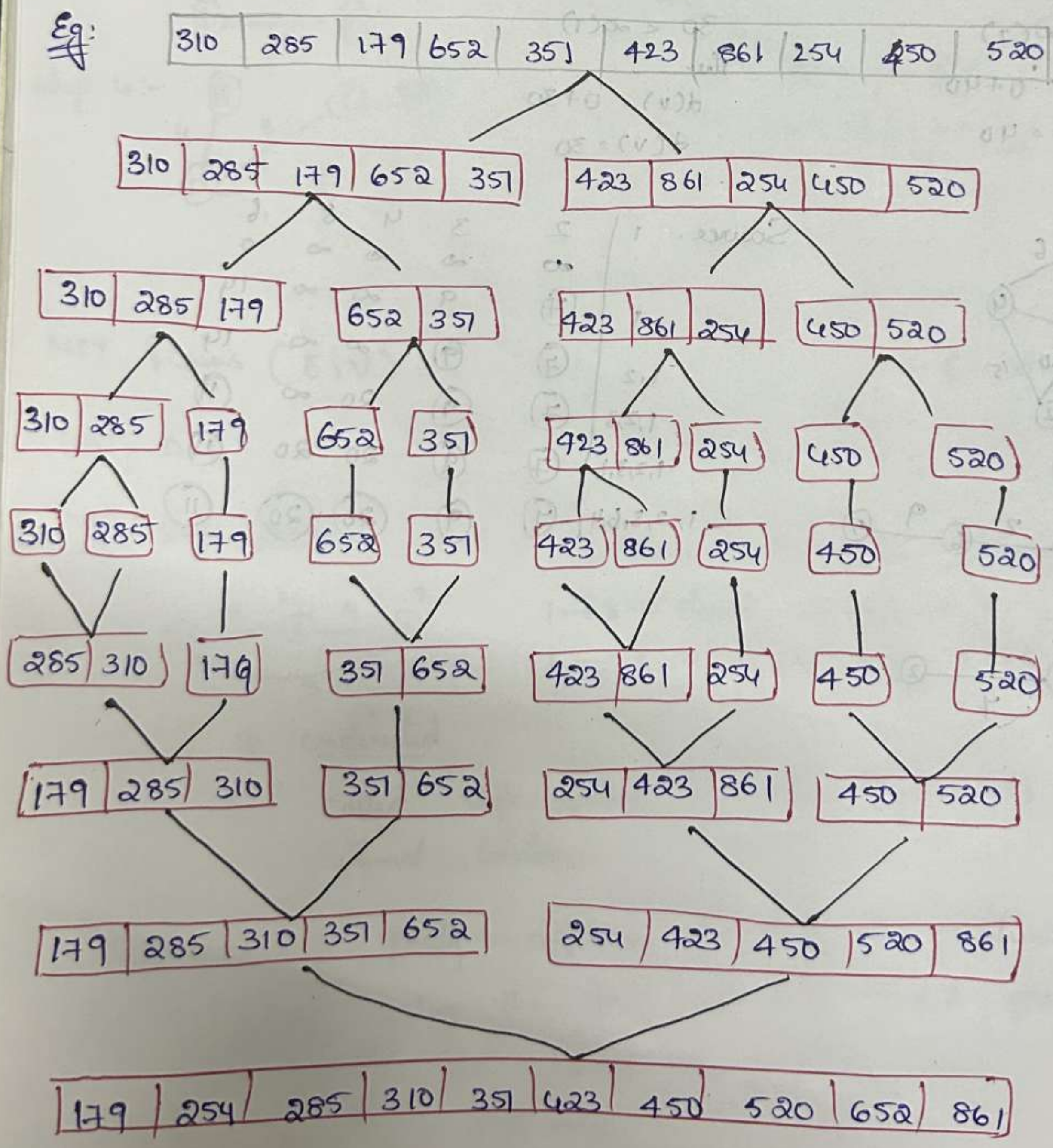
Best, Avg, Worst

Merge Sort

$WC = O(n \log_2 n)$

- Each set is individually sorted.
- Given sequence of 'n' elements is divided into 2 sets $a[1] \ldots a[\lfloor n/2 \rfloor]$ and $a[\lfloor n/2 \rfloor + 1] \ldots a[n]$.
- Resulting sorted sequences are merged to produce a single sorted sequence of 'n' element.

Eg:



Algorithm using Recursion

```
Algo Mergesort(low, high)
{
 if (low<high) then
 {
   mid := ⌊(low+high)/2⌋;
   Mergesort (low, mid);
   Mergesort (mid+1, high);
   Merge (low, mid, high);
 }
}
```

Merging 2 Sorted Sub-array using auxiliary storage:

```
Algo Merge (low, mid, high)
{
 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;
 }
```

Sorted

for k := low to high do

a[k] := b[k];

}

Time Complexity

Recurrence Relation

$$T(n) = \begin{cases} 1 & , n=1 \\ T(n/2) + T(n/2) + cn, & n>1 \end{cases}$$, c is constant

$T(n) = 2T(n/2) + cn$ (or)

$T(n) = a \cdot T(n/b) + f(n)$

no. of sub arrays — no. of elements in the sub-array

$n = 2^k$

(using backward substitution)

$$\begin{aligned}
T(n) &= 2 \cdot T(n/2) + n \\
&= 2\left[2T(n/4) + n/2\right] + n \\
&= 4T(n/4) + \cancel{2} \cdot n/\cancel{2} + n \\
&= 4T(n/4) + 2n \\
&= 4\left[2T(n/8) + n/4\right] + 2n \\
&= 8T(n/8) + 4 \cdot n/4 + 2n \\
&= 8T(n/8) + 3n \\
&= 2^3 T(n/2^3) + 3n \\
&= 2^k T(n/2^k) + kn \\
&= n \cdot T(n/n) + \log_2 n \cdot n
\end{aligned}$$

$$\boxed{T(n) = O(n \log_2 n)}$$

$T(n) = n \cdot T(1) + n\log_2 n$

Quick Sort (Divide & Conquer Approach)

- used to arrive at an efficient sorting method different from merge sort.
- In quicksort, the division into 2 sub-arrays is made so that the sorted sub-arrays do not need to be merged later.

  - Divide
  - Conquer - Recursively sort the 2 sub-arrays.
  - Combine

left sub-array	P	Righ Sub-array
$\underbrace{\quad}_{A[0] \,---\, A[m-1]}$	$\uparrow$ pivot	$\underbrace{\quad}_{A[m+1] \,--\, A[n-1]}$

Pivot Element:

Pivot element can be choosen in different ways:

- 1^{st} element
- Last element
- Random element

Procedure

- Given array consists of n elements.
- If array consists of only 1 element then return results, else
  - → Pick 1 element to choose pivot
  - → partitioning elements into 2 sub-arrays
- If $(a[i] \leq pivot)$ then i++, otherwise stop & increment i
- If $(a[j] \geq pivot)$ then j--, other stop decrement j
- If both conditions are failed then exchange a[i] with a[j].
- If a[j] < a[pivot] & j has crossed i, i.e, j < i then swap/ Exchange pivot element with a[j].

Example

Pivot

50 30 10 90 80 20 40 70

↑ i ↑ j

1) 50 30 10 90 80 20 40 70

pivot i j

2) 50 30 10 90 80 20 40 70

pivot i j

3) 50 30 10 90 80 20 40 70

pivot i j

4) 50 30 10 40 80 20 90 70

pivot i j

5) 50 30 10 40 80 20 90 70

pivot i j

6) 50 30 10 40 20 80 90 70

pivot j i

In above step a[j] < pivot and j has crossed with i

i.e., j < i then we will swap pivot with a[j]

7) 20 30 10 40 50 80 90 70

left sub-list ↓ j right-sub list

pivot

Now we will sort left sub-list, assuming 1st element of left Sub-list as pivot.

Now pivot = 20

8) 20 30 10 40 50 80 90 70

pivot i j ↓

occupied its position

9) 20 30 10 40 50 80 90 70

pivot i j

10) 20 10 30 40 50 80 90 70
pivot i j

11) 20 10 30 40 50 80 90 70
pivot j i

12) 10 20 30 40 50 80 90 70
left ↑ pivot Right

Considering right Sub-list

13) 10 20 30 40 50 80 90 70
pivot i j

14) 10 20 30 40 50 80 70 90
pivot j i

15) 10 20 30 40 50 70 80 90
↑ pivot

Final Sorted list:

10 20 30 40 50 70 80 90

<u>Algorithm</u>

- Quick
- Partition

```
Algo Quicksort (P, q)
{
 if(P<q) then
 {
  j: = Partition (a, P, q+1);
  Quicksort (P, j-1);
  Quicksort (j+1, q);
 }
}
```

<u>Partition</u>

```
Algorithm Partition(a, m, P)
{
  pivot:= a[m];
  i: = m;
  j: = P;
  repeat
  {
  repeat
     i: = i+1;
  until (a[i] ≥ pivot);
  repeat
     j: = j-1;
  until (a[j] ≤ pivot);
```

```
    if (i<j) then interchange (a, i, j);
  }
  until (i≥j);
  a[m]: = a[j];
  a[j]: = pivot;
  return j;
}
```

Interchange Algorithm

```
Algorithm Interchange (a, i, j)
{
  temp p;
  p: = a[i];
  a[i]: = a[j];
  a[j]: = p;
}
```

Performance Analysis

BC Time Complexity - Split in the middle

Recurrence relation:

$$T(n) = \begin{cases} 1 & , n=1 \\ T(n/2) + T(n/2) + n & ; n>1 \end{cases}$$

$$\therefore T(n) = 2T(n/2) + n$$

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

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

$$= 4[2T(n/8) + n/4] + 2n$$

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

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

$$= 2^3 T(n/2^3) + 3 \cdot n$$

$\vdots$

$= 2^k T(n/2^k) + k.n$

let $2^k = n$

Apply log on both sides

$\log 2^k = \log n$

$k \log 2 = \log n$

$k = \log n$

$T(n) = n.\,T(n/n) + \log n . n$

$= n.\,T(1) + n.\log n$

$= n.(1) + n \log n$

$= n + n\log n$

BC $\boxed{T(n) = O(n \log n)}$

worst case

when pivot is min or max

time requires

n - - - - - - - → n

1 n-1 - - - - - - → n-1

2 n-2 - - - - → n-2

3 n-3 - - - - → n-3

⋮ - - - → 1

$$T(n) = \begin{cases} 1 & , n=1 \\ T(n-1)+cn & , n>1 \end{cases}$$

$T(n) = T(n-1) + n$ ($\because c = $ constant)

$= n + (n-1) + (n-2) \cdots + 2 + 1$

As WKT

$1+2+3 \cdots + n = \frac{n(n+1)}{2}$

$= \frac{n^2+n}{2}$

$\boxed{\therefore T(n) = O(n^2)}$

Average case

Same as best case

$O(n \log n)$

Iterative Version of Quicksort

```
Algorithm Quicksort (P, q)
{
 repeat
 {
  while (P<q) do
  {
   j: = Partition (a, P, q+1);
   if ((j-P) < (q-j)) then
   {
    Add(j+1);
    Add(q);
    q: =j-1;
   }
   else
   {
    Add(P);
    Add(j-1);
    P: =j+1;
   }
  }
  if Stack is empty then return;
   Delete (P);
   Delete (q);
 } until (false);
}
```

$$S(n) = \begin{cases} 0 & , n \leq 1 \\ 2 + S(\lfloor (n-1)/2 \rfloor) & , n > 1 \end{cases}$$

which is less than $2\log n$

Strassen's Matrix Multiplication

- 2 matrices A and B of size $n*n$

$C = A*B$ is also $n*n$ matrix.

$$c(i,j) = \sum_{1 \le k \le n} A(i,k)\, B(k,j) \text{ for all } i \,\&\, j \text{ between } 1 \,\&\, n.$$

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

Algorithm for matrix multiplication:

```
Algo MM(A, B, C)
{
for i:=1 to n do                         ----> n+1
 {
 for j:=1 to n do                        ----> n(n+1)
  {
   C[i,j]:=0;                            ----> n*n
   for k:=1 to n do                      ----> n*n*(n+1)
    {
    C[i,j]:= A[i,k]*B[k,j];              ----> n*n*n
    }
  }
 }
}
```

$$\boxed{T(C) = O(n^3)}$$

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

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

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

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

Applicable for $2*2$ matrix

If $n > 2$, the elements of c can be computed using matrix multiplication and addition operations applied to matrices of size $n/2 * n/2$

This algorithm will continue applying itself to smaller sized sub-matrices until 'n' becomes suitably small ($n = 2$) so that the product is computed directly.

Time Complexity:

- To compute $A*B$ – 8 multiplications of $n/2 * n/2$ & 4 additions.
- Since 2 $n/2 * n/2$ matrices can be added in time Cn^2 ($\because c$ = constant)

Overall computing time $T(n)$ of resulting Divide & Conquer algorithm by the recurrence relation:

$$T(n) = \begin{cases} 1 & , n \le 2 \\ 8T(n/2) + n^2 & , n > 2 \end{cases}$$

$a = 8, b = 2, f(n) = n^2$ and $n^k = n^2$

$$\boxed{\log_b^a = \log_2^8 = 3}$$

$$\boxed{T(n) = O(n^3)}$$

- Volker Strassen's has discovered a way to compute C_{ij}'s of using only 7 multiplication & 18 additions or Substractions
- Involves, 1st computing seven $n/2 * n/2$ matrices P, Q, R, S, T, U & V then C_{ij}'s are computed.
- As it can be seen P, Q, R, S, T, U & V can be computed using 7 matrix multiplications & 10 matrix additions or Substractions.
- The C_{ij}'s require an additional 8 additions / Subtractions

$$
\boxed{
\begin{aligned}
P &= (A_{11} + A_{22})(B_{11} + B_{22}) \\
Q &= (A_{21} + A_{22})B_{11} \\
R &= A_{11}(B_{12} - B_{22}) \\
S &= A_{22}(B_{21} - B_{11}) \\
T &= (A_{11} + A_{12})B_{22} \\
U &= (A_{21} - A_{11})(B_{11} + B_{12}) \\
V &= (A_{12} - A_{22})(B_{21} + B_{22}) \\
\\
C_{11} &= P + S - T + V \\
C_{12} &= R + T \\
C_{21} &= Q + S \\
C_{22} &= P + R - Q + U
\end{aligned}
}
$$

Now, we will compare the actual our traditional matrix multiplication procedure with Strassen's procedure

In Strassen's multiplication

$$
\begin{aligned}
C_{11} &= P + S - T + V \\
&= (A_{11} + A_{22})(B_{11} + B_{22}) + A_{22}(B_{21} - B_{11}) - (A_{11} + A_{12})B_{22} + (A_{12} - A_{22})(B_{21} + B_{22}) \\
&= A_{11}B_{11} + \cancel{A_{11}B_{22}} + \cancel{A_{22}B_{11}} + \cancel{A_{22}B_{22}} + \cancel{A_{22}B_{21}} - \cancel{A_{22}B_{11}} - \cancel{A_{11}B_{22}} - \cancel{A_{12}B_{22}} + A_{12}B_{21} + \\
&\quad \cancel{A_{12}B_{22}} - \cancel{A_{22}B_{21}} - \cancel{A_{22}B_{22}}
\end{aligned}
$$

$$\therefore \boxed{C_1 = A_{11}B_{11} + A_{12}B_{21}}$$

Tc for Strassen's matrix multiplication:

Totally – 7 matrix multiplications & 10 additions / subtractions

Recurrence relation for $T(n)$

$$
T(n) = \begin{cases} 1 & , n \leq 2 \\ 7T(n/2) + n^2 & , n > 2 \end{cases}
$$

$a = 7 \quad b = 2 \quad f(n) = n^2 \quad n^k = n^2$

$$\boxed{\log_b a = \log_2 7 = 2.81}$$

$$\boxed{T(n) = O(n^{2.81})}$$

Hence, compared to traditional matrix multiplication TC ($O(n^3)$) in Strassen's matrix multiplication TC will be reduced i.e., $O(n^{2.81})$

Eg. $A = \begin{bmatrix} -1 & 0 & 2 & 3 \\ 9 & 5 & 5 & 6 \\ 4 & 7 & 8 & 2 \\ 7 & -2 & 3 & -9 \end{bmatrix}$ (A_{11}, A_{12}, A_{21}, A_{22}) $\quad B = \begin{bmatrix} -7 & 2 & 1 & 4 \\ -9 & 6 & 7 & 8 \\ -1 & 0 & -4 & 3 \\ -8 & 1 & 5 & -6 \end{bmatrix}$ (B_{11}, B_{12}, B_{21}, B_{22})

C = ?

MaxMin (Divide & Conquer technique)

* Problem is used to find the maximum and minimum items in a set of 'n' elements.

Algorithm : Straight forward Maximum and Minimum

```
Algorithm StraightMaxMin (a, n, max, min)
{
  max: = min: = a[1];
  for i: = 2 to n do
  {
    if (a[i]>max) then max: = a[i];
    if (a[i]<min) then min: = a[i];
  }
}
```

Recursive Algorithm - Maximum and minimum

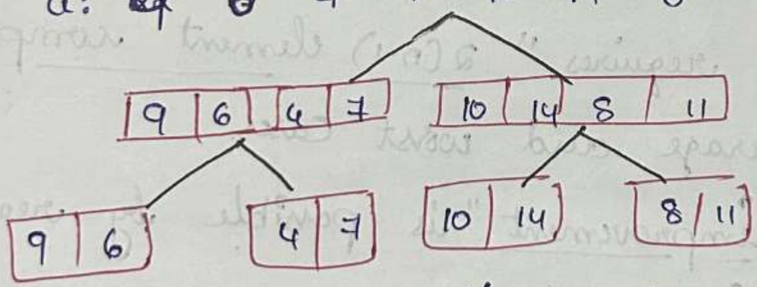
```
Algorithm MaxMin (i, j, max, min)
{
  if (i==j) then max: = min: = a[i];
  else if (i==j-1) then
  {
    if (a[i]<a[j]) then
    { max: = a[j]; min: = a[i];
```

```
}
else
{
 max: = a[i]; min: = a[j];
}
}
else
{
 mid: = ⌊(i+j)/2⌋;
 MaxMin (i, mid, max, min);
 MaxMin(mid+1, j, max1, min1);

 if (max < max1) then max: = max1;
 if (min > min1) then min: = min1;

}
}
```

Eg: a: 9 6 4 7 10 14 8 11



max:	9	7	14	11
min:	6	4	10	8

max:	9	14
min:	4	8

max: 14
min: 4

* Analyzing the time complexity of this algorithms, we once again concentrate on the "number of element comparisons"
* The justification for this is that the frequency count for other operations in this algorithm is of the same order as that for element comparisons.
* More, importantly, when the elements in a[1:n] are Polynomials, Vectors, Very large numbers, or strings of characters the cost of an element comparison is much higher than the cost of the other operation
* Hence the "time is determined mainly by the total cost of the element comparisons."
* "Straight MaxMin" requires "2(n-1) element comparisons" in the best, average and worst cases.
* An "immediate improvement" is possible by realizing that the comparison "a[i] < min" is necessary only when "a[i] > max" is false.

Hence we can replace the contents of the for loop by:

" if (a[i] > max) then max := a[i];
else if (a[i] < min) then min := a[i]; "
* Now the "best case" Occurs when the elements are in "Increasing order" the no. of element comparisons is n-1.

* The "Worst Case" occurs when the elements are in "decreasing order". in this case the no. of elements comparisons is "$2(n-1)$".

* A "divide - and conquer" algorithm for this problem 28 would proceed as follows:

→ Let $P = (n, a[i], \dots a[j])$ denote an arbitrary instance of the problem.

→ Here n is the no. of elements in the list, $a[i] \dots a[j]$ and we are intersted in finding the maximum and mimimum of this lost

→ Let Small (P) / be true when $n \le 2$ In thos case, the maximum and minimum are $a[i]$ if $n=1$ if $n=2$, the problem can be solved by making one comparison

* If the list has more than two eliments, has to be divided into smaller instances

for example :- We might P divided into two instances

$$P_1 = ([n/2], a[1], \dots a([n/2])) \text{ and}$$

$$P_2 = (n - [n/2], a[(n/2] + 1, \dots, a[n].$$

* After having divided P into two smaller sub problems we can solve them by recursively invoking the same divide - and conquer algorithm.

* How can we combine the solutions for P_1 and P_2 to obtain a solution for P ?

→ If (Max (P) and Min(P) are the maximum and minimum of the elements in P_1 then

" Max (P) is the larger of Max (P_1) and Max (P_2)."

→ And also " Min (P) is the Smaller of Min (P_1) and Min (P_2) "

Time Complexity :-

Find the no. of element comparisons needed for MaxMin? if T(n) represents this number, then the resulting recurrence relation is

$$T(n) = \begin{cases} T(\lceil n/2 \rceil) + (\lceil n/2 \rceil + 2 & , n > 2 \\ 1 & , n = 2 \\ 0 & , n = 1 \end{cases}$$

When n is a power of two, $n = 2^k$ for some positive integer k, then $T(n) = 2T(n/2) + 2$

$$\begin{aligned} \Rightarrow T(n) &= 2T(n/2) + 2 \\ &= 2(2T(n/4) + 2) + 2 \\ &= 4T(n/4) + 4 + 2 \\ &= 4[2T(n/8) + 2] + 4 + 2 \\ &= 8T(n/8) + 8 + 4 + 2 \\ &= 2^3 . T(n/2^3) + 2^3 + 2^2 + 2^1 \\ &= 2^k . T(n/2^k) + \Sigma_{1 \le i \le k-1} 2^i \\ &= n . T(n/k) + 2^k - 2 \quad (\because n = 2^k) \\ &= n . T(1) + n - 2 \\ &= n . 2 + n - 2 \end{aligned}$$

$$\boxed{T(n) = 3n - 2}$$

* The time complexity of recursive algorithm to find maximum and minimum is : <u>O(n)</u>