Intro to AOA

Q.1 What is asymptotic Analysis ? Define Big O, Omega and Theta Notations.

⇒ • **Assymptotic Notation**

⇒ Asymptotic analysis are a mathematical tool to find time or space complexity of an algorithm without implementing it in a programming language.

- It is a way of describing a major component of the cost of the entire algorithm.
- This is measure independent of machine specific constants.

• Big O

⇒ It is denoted by 'O', and it is pronounced as 'big Oh'.

- Big O notation defines the upper bound for the algorithm it means the running time of algorithm cannot be more than its asymptotic upper bound for any random sequence of data.
- Let f(n) and g(n) are two non-negative functions indicating running time of two algorithms.
- We say g(n) is the upper bound of f(n) if there exists some positive constants c and n_0 such that $0 \leq f(n) \leq c \times g(n)$ for all $n \geq n_0$.
- It is denoted by $f(n) = O(g(n))$.

• Big Omega

⇒ It is denoted by 'Ω' and it is pronounced as "big omega".

- Big Omega defines the lower bound for the algorithm it means the running time of algorithm cannot be less than its asymptotic lower bound for any random sequence of data.
- Let f(n) and g(n) are two non-negative functions indicating running time of algorithm.
- We say g(n) is the lower bound of function f(n) if there exist some positive constants c and n_0 such that $0 \leq c \cdot g(n) \leq f(n)$ for all $n \geq n_0$.
- It is denoted by $f(n) = \Omega(g(n))$.

• Big Theta

⇒ It is denoted by 'θ' and it is pronounced as "Big Theta".

- Big Theta defines the tight bound for the algorithm it means running time of algorithm cannot be less than or greater than it's asymptotic tight bound for any random sequence of data.
- Let f(n) and g(n) are two non-negative functions indicating running time of two algorithms.
- We say g(n) is tight bound of f(n)

if there exist some +ve constants c_1, c_2, and n_0 such that
$0 \leq c \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for all $n \geq n_0$.

- It is denoted by -
$f(n) = \Theta(g(n))$.

3.2 Short Note on P, NP, NP Complete.

⇒

- **P Problems**

⇒ P problems are set of problems that can be solved in polynomial time by deterministic algorithms.
- P is also known as PTIME or DTIME complexity class.
- Problems which can be solved in polynomial time, which take time like $O(n)$, $O(n^2)$, $O(n^3)$.
- For example, finding maximum element from an array or to check whether string is palindrome or not
- It excludes all the problems which cannot be solved in polynomial time.
- Knapsack problem using brute force cannot be solved in polynomial time. Hence, it is not P problem.
- E.g.,
  Insertion Sort,
  Merge Sort,
  Linear Search
  Matrix multiplication.

• NP Problem

⇒ NP is set of problems which can be solved in non-deterministic polynomial time.

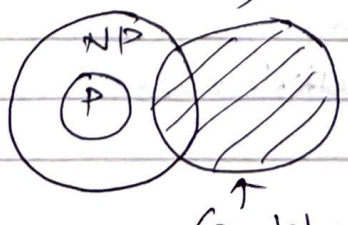
- NP does not mean non-polynomial, it stands for Non-deterministic polynomial time.
- the non-deterministic operates in two stages.
- i) Non-deterministic stage (guessing)
  ii) Deterministic stage. (verification)
- Solution to the NP problems cannot be obtained in polynomial time, but given the solution, it can be verified in polynomial time.
- For e.g., Knapsack problem, TSP, etc.

• NP-Complete

⇒ The group of problem which are in both NP and NP-Hard are known as NP-Complete problem.

- Decision problem C is called as NP complete if it has following two properties:

i) C is in NP, and
ii) Every problem X in NP is reducible to C in polynomial time, i.e For every $X \in NP$, $X \leq_p C$.



Q.3 Differentiate b/w P and NP.

⇒

P Problem	NP Problem
i) P stands for polynomial.	i) NP stands for non-deterministic polynomial.
ii) Problem can be solved in polynomial time.	ii) Problem cannot be solved in polynomial time, but given the solution it can be verified in polynomial time.
iii) P problems are subset of NP problem.	iii) NP problems are superset of P problem.
iv) All P problems are deterministic in nature.	iv) All NP problem are non-deterministic in nature.
v) Eg., Insertion sort, Merge sort, etc.	v) Eg., TSP, Knapsack.

Q.4 Explain recurrence and various method to solve recurrences.

⇒ • Recurrence

⇒ Recurrence equation recursively defines a sequence of function with different argument ; behaviour of recursive algorithm ~~are~~ is better represented using recurrence equations.

- Recurrence are normally of the form-

$$T(n) = T(n-1) + f(n), \text{ for } n > 1$$
$$T(n) = 0, \text{ for } n = 0$$

- The function $f(n)$ may represented ~~in~~ constant or any polynomial in n.
- $T(n)$ is represented as time required to solve problem of size n.
- Recurrence of linear search
  ⇒ $T(n) = T(n-1) + 1$.

1) Substitution Method

⇒ Linear homogenous recurrence of polynomial order greater than 2 hardly arises in practice.

- It is also known as iteration method.
- There are two ways to solve such equations :

1) Forward Substitution

⇒ This method finds the solution of the smaller problem using base condition.

For eg:)
Recurrence of linear search is,
$T(n) = T(n-1) + 1$ -(1), for $n > 0$ and
$T(n) = 0$, for $n = 0$ (base cond^n)
$\Rightarrow T(0) = 0$ -(2)

Putting $n = 1$ in eq (1),

$T(1) = T(0) + 1 = 0 + 1 = 1$

Putting $n = 2$ in eq (2),

$T(2) = T(1) + 1 = 1 + 1 = 2$

After k iterations,

$T(k) = k$

And for $k = n$,

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

2) Backward substitution

⇒ This method substitutes value of n by $n-1$ recursively, to solve smaller and smaller problem.

For e.g.,

$T(n) = T(n-1) + n$ -(1)

Replace, n by $n-1$ in (1),

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

Put (2) in (1),

$T(n) = [T(n-2) + (n-1)] + n$ -(3)

Replace n by $n-1$ in eq (2),

$T(n-2) = T(n-3) + (n-2)$ -(4)

$\therefore T(n) = T(n-3) + (n-2) + (n-1) + n$

After k iterations,

$T(n) = T(n-k) + (n-k+1) + \cdots (n-1) + n$

$T(n) = \sum n$

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

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

2) Recurrence Tree :

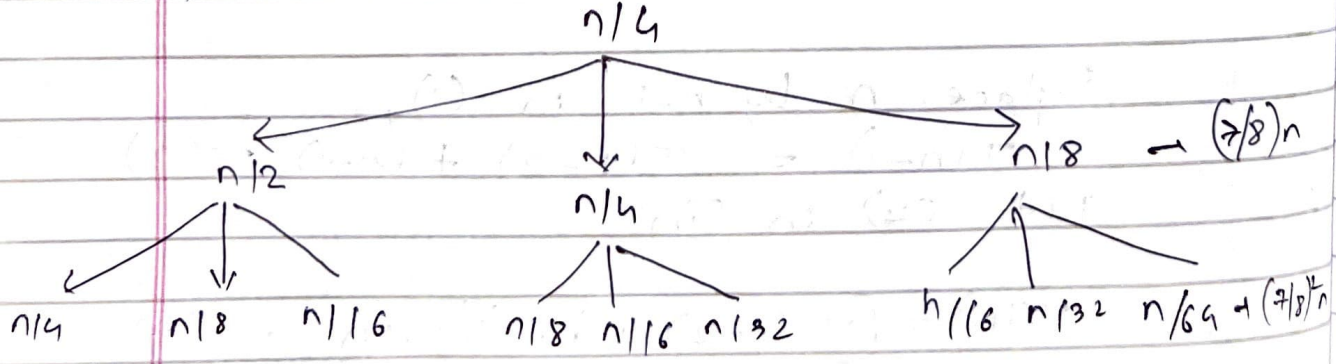
⇒ Recurrence tree method provides effective and yet simple way of solving the recurrences.

- Ultimately, recurrence is set of functions, each branch in recurrence tree represents the cost of solving one problem from the family of problems belonging to given recurrence.

For e.g.)

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

⇒



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

$$T(n) \leq n + (7/8)n + (7/8)^2 n + \cdots + (7/8)^h n$$

$$= n \sum_{i=0}^{h} \left(\frac{7}{8}\right)^i$$

$$= n \times \frac{1}{1 - 7/8}$$

$$= 8n$$

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

3) Master Method

⇒ Master method is used to quickly solve the recurrence of the form $T(n) = a \cdot T(n/b) + F(n)$.

- Master method finds the solution without substituting the values of $T(n)$.
- In above equation -
  - n = size of problem
  - a = No. of sum problems created in
  - n/b = size of each sub problem
  - $f(n)$ = Work done outside recursive call.
- If $f(n)$ is polynomial of degree d and
  1. $T(n) = \Theta(n^d \log n)$ If $a = b^d$
  2. $T(n) = \Theta(n^{\log_b a})$ If $a > b^d$
  3. $T(n) = \Theta(n^d)$ If $a < b^d$

For e.g.,

$$T(n) = 8T\left(\frac{n}{2}\right) + n^2$$

$\Rightarrow$

Here $b = 2$, $a = 8$, $f(n) = n^2$, $d = 2$

As,

$a > b$ i.e $8 > 2^2$

$$\therefore T(n) = \theta(n^{\log_b a})$$
$$= \theta(n^{\log_2 8})$$
$$= \theta(n^3) \quad //$$

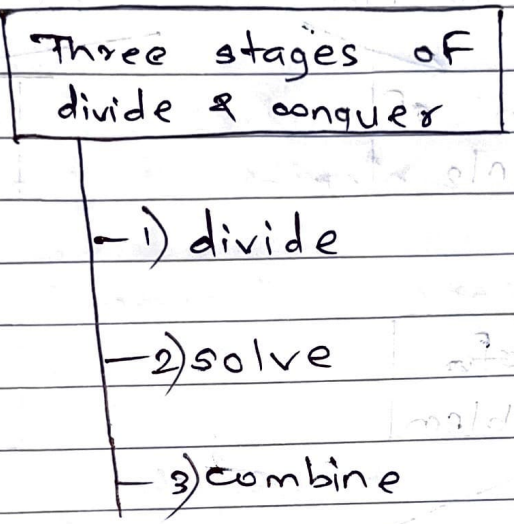
Divide and Conquer

8.1 Explain the general procedure of divide and conquer method.

⇒ i) Divide and conquer

⇒ This is the most widely applicable technique for designing efficient algorithms.

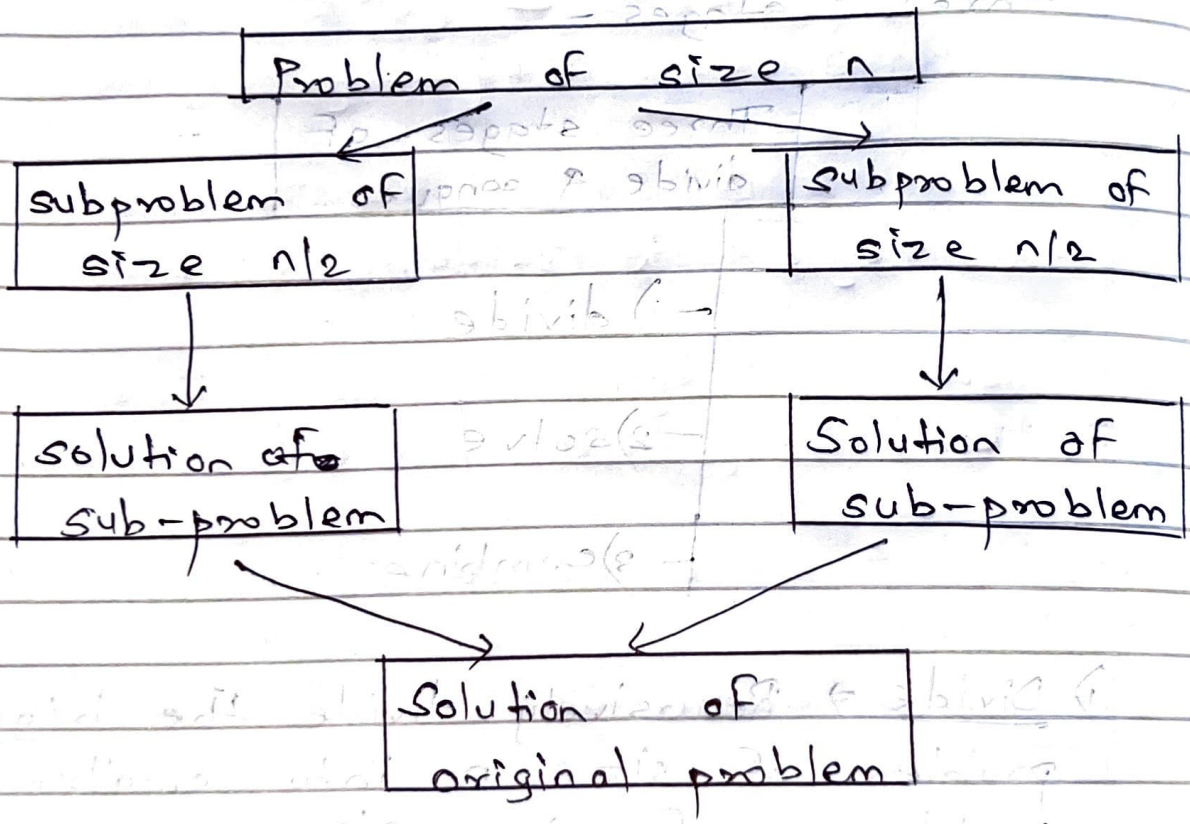
- Divide and conquer operates in three stages -



1) Divide ⇒ Recursively divide the bigger problem of size n into smaller sub-problems of size n/2.
2) Solve ⇒ Subproblems are solve independently.
3) Combine ⇒ Combines the solution of subproblems in order to derives the solution of original big problem.

- Subproblems are similar to the original problem with smaller arguments, hence such problems can be easily solved using recursion.
- Divide and conquer is multi-branched,

- Top down recursive approach
- Each branch indicates one subproblem and it calls itself with smaller argument.
- Graphical representation of divide and conquer -



- Sub-problems may or may not be of size n/2.

Q.3 Write algorithm of merge sort.

⇒
- Two-way merge is the process of merging two-sorted arrays of size m and n into a single sorted array of size (m+n).
- Algorithm :

```
MERGE_SORT (A, beg, end)
{
    if (beg < end) do  mid = (beg+end)/2
        MERGE_SORT (A, beg, mid),
        MERGE_SORT (A, mid+1, end)
        COMBINE (A, beg, mid, end)
}

COMBINE (A, beg, mid, end)
{
    n1 = mid - beg + 1
    n2 = end - mid

    for i ← 1 to n1 do
        LEFTARR [i] ← A [beg + i]

    for j ← 1 to n2 do
        RIGHTARR [j] ← A [mid + j + 1]

LEFTARR [n1 + 1] ← ∞
RIGHTARR [n2 + 1] ← ∞

    i ← 1 , j ← 1
```

$T = 2T\left(\frac{n}{2}\right) + n$

```
for K ← beg to end do
    if LEFTARR[i] ≤ RIGHTARR[j]
        A[K] ← LEFT[i]
        i ← i+1
    else
        A[K] ← RIGHT[j]
        j ← j+1
}
```

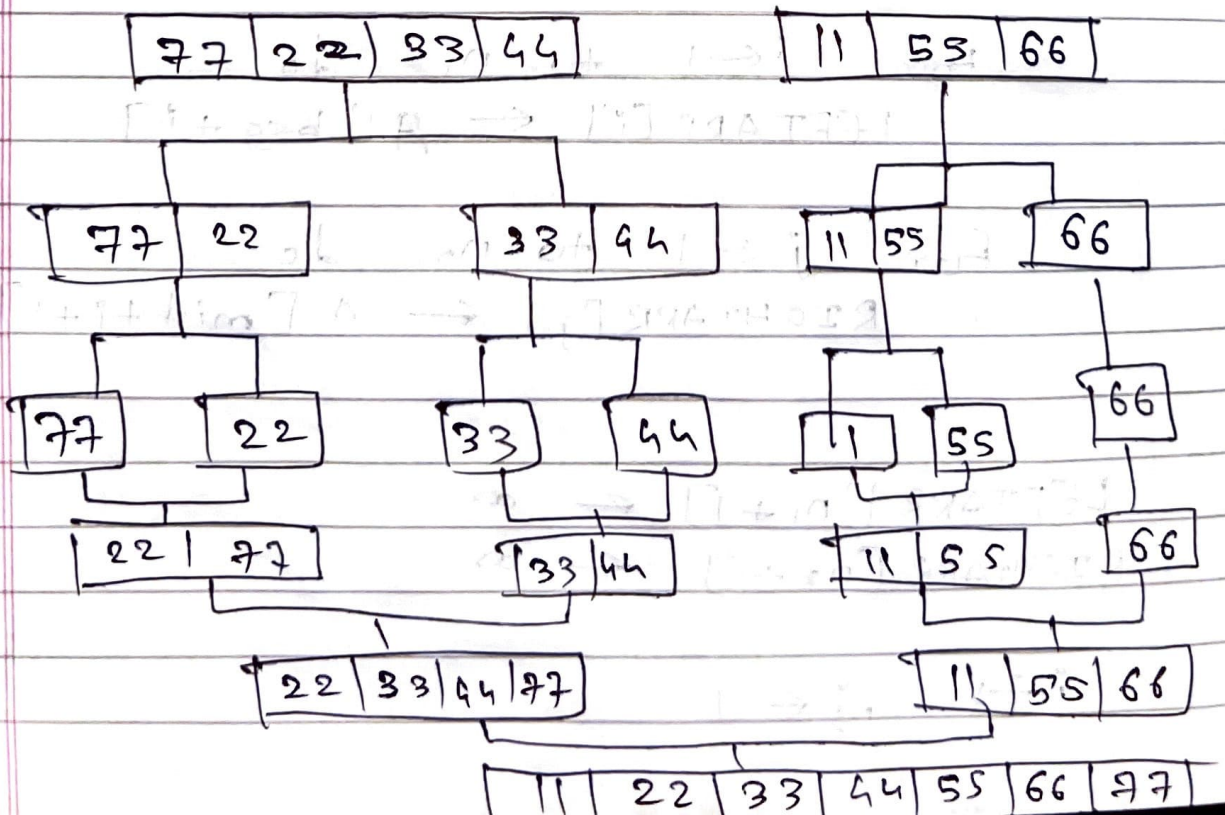
- Time complexity of Merge sort

⇒

Best Case	Average Case	Worst Case
$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n \cdot \log n)$

For e.g.) <77, 22, 33, 44, 11, 55, 66>

i) It will divide above array two sub-arrays.



Q.3 Quick Sort

⇒ • Algorithm :

```
QUICKSORT (A, beg, end)
{
    if beg ≤ end do
        q ← PARTITION (A, beg, end)

        QUICKSORT (A, beg, q-1)
        QUICKSORT (A, q+1, end)
}

PARTITION (A, beg, end)
{
    x ← A[end]
    i ← beg - 1

    for j ← beg to end-1 do

        if A[j] ≤ x then
            i ← i+1
            swap (A[i], A[j])

    swap (A[i+1], A[end])

    return (i+1)
}
```

- Complexity Analysis:

1) Worst case

⇒ The coorst-case behaviour occurs when partitioning routine produces one subproblem with n-1 and one with 0 elements.

The partitioning costs $\Theta(n)$.

$$T(n) = T(n-1) + \Theta(n) + \Theta(1)$$
$$= T(n-1) + n$$
$$\therefore T(n) = O(n^2)$$

2) Average Case

⇒ The average-case running time of quick sort is much closer to the best case.

$$T(n) = O(n \cdot \log_2 n)$$

3) Best Case

⇒ In most even posible split, PARTITION produces two subproblems, each of size no more than $n/2$, since one is of size $(n/2)$ and one of size $(n/2) - 1$.

$$T(n) \leq 2T(n/2) + \Theta(n)$$

$$\therefore T(n) = O(n \cdot \log_2 n)$$

Binary search

⇒ i) Searching is the problem of finding an element from given set of data.

ii) Binary search is divide and conquer approach.

iii) Binary search is efficient than linear search.

iv) For binary search, array must be sorted, which is not required in linear search.

Algorithm

⇒ B (A, key) {

low ← 1

high ← n

while low < high do

mid ← (low+high)/2

if A[mid] == key then

return mid

else if A[mid] < key then

low ← mid+1

else

high ← mid-1

}

- Complexity :

BEST ⇒ O(1)

Average ⇒ O(n.logn) ⇒ worst ⇒ O(n·logn)

Q. Strassen's Matrix Multiplication.

⇒ i) Strassen has proposed divide and conquer strategy based algorithm, which takes less numbers of multiplication compares to this traditional way of matrix multiplication.

ii) Using strassen's method, multiplication operation is defined as -

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

$C_{11} = S_1 + S_4 - S_5 + S_7$

$C_{12} = S_3 + S_5$

$C_{21} = S_2 + S_4$

$C_{22} = S_1 + S_3 - S_2 + S_6$

Where,

$S_1 = (A_{11} + A_{22}) * (B_{11} + B_{22})$

$S_2 = (A_{21} + A_{22}) * B_{11}$

$S_3 = A_{11} * (B_{12} - B_{22})$

$S_4 = A_{22} * (B_{21} - B_{11})$

$S_5 = (A_{11} + A_{12}) * B_{22}$

$S_6 = (A_{21} - A_{11}) * (B_{11} + B_{12})$

$S_7 = (A_{12} - A_{22}) * (B_{21} + B_{22})$

- Let us check if it is same as conventional approach.

$C_{12} = S_3 + S_5$

$\therefore C_{12} = A_{11} * (B_{12} - B_{22}) + (A_{11} + A_{12}) * B_{22}$

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

- This is same as C_{12} derived using conventional approach.

- Complexity Analysis :

Recurrence relation is given by -

$T(n) = 7 \cdot T(n/2)$.

Put $n = n/2$

$T(n/2) = 7\,T(n/4)$

$\therefore T(n) = 7^2 \cdot T(n/4)$

$\vdots$

$\therefore T(n) = 7^k \cdot T(n/2^k)$

Let $n = 2^k \Rightarrow k = \log_2 n$

$T(2^k) = 7^k T(1)$

$= 7^k$

$= 7^{\log_2 n}$

$= n^{\log_2 7}$

$= n^{2.81}$

$\approx n^3$ //

$T(n) = O(n^{2.81})$.