1/6/23

D.V.K

Design and Analysis of Algorithms

UNIT-I :- Introduction to Algorithms

Fundamentals of algorithmic problem solving - Analysis framework - performance Analysis :- space complexity - Time complexity - Growth of functions : Asymptotic Notation - Big oh notation, omega notation, theta notation, little oh.

Algorithm :- An Algorithm is a (setep by step) procedure sequence of computational steps to solve any prob for solving a problem (or) to complete a task.

pseudocode :- pseudocode is an informal user friendly and artificial language that helps programmers to develop an Algorithm

man made

Algorithm which is written in english like language is known as pseudo code.

Example :- Addition of two numbers

```
① start
② Read two values
③ Add two values
④ store result in another variable
⑤ print the variable which contains result
⑥ stop
```

(or)

```
① start
② Read a, b
③ Add a, b
④ store result in variable c
⑤ print c
⑥ stop.
```

Ex:- Average of 3 - numbers

① start

② Read 3 values, Assume a, b, c

③ sum a, b, c

④ Divide sum by 3

⑤ store result in another variable d

⑥ print "d"

⑦ stop (or) end program.

characteristics (or) properties of an Algorithm :-

① Input / output :-

Each Algorithm must take 0, 1 (or) more quantities as input and should produce atleast one output.

② finiteness :- An Algorithm should terminate in a finite no. of steps.

③ Definiteness (or) unambiguous

Each step of algorithm must be clear means no ambiguity

④ Effectiveness :-

Algorithm should contain only necessary steps, it should not contain any unnecessary statements.

⑤ Generality :-

Algorithm should run for any type of input or data

performance and Analysis of Algorithm :-

An algorithm is said to be efficient and fast, if it takes less time to execute and consume less memory space.

In order to analyze performance of an algorithm we use ① Time Complexity
② Space Complexity.

Time Complexity :-

the amount of time that an algorithm requires for its execution is known as time complexity

① **Best Case time Complexity :-**

If an Algorithm requires minimum amount of time for its execution then it is known as Best case time Complexity.

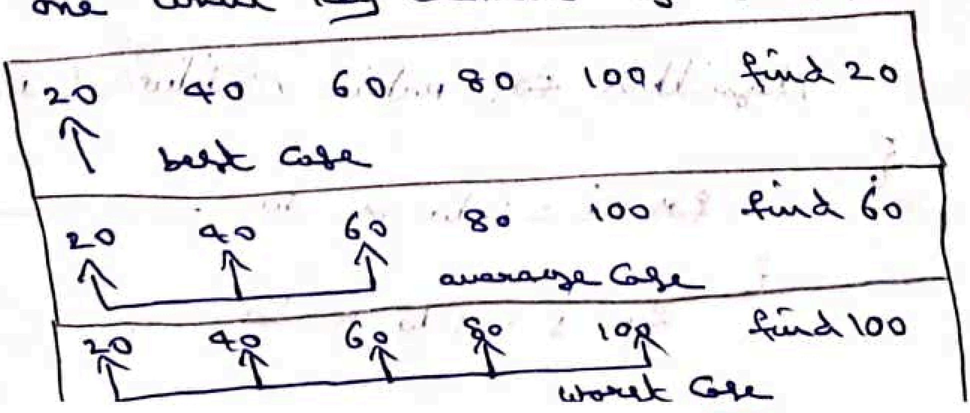
② **worst case time complexity :-**

If an Algorithm requires maximum amount of time for its execution then it is known as worst case time complexity.

③ **Average Case time Complexity :-**

If an Algorithm requires average amount of time for its execution then it is known as average time complexity

Ex1:- Linear search :- It is searching an element one by one until key element is found.



pseudo code approach :-

```
Algorithm Even or odd (n)
{
        If (number % 2 == 0) then
            print even;
        else
            print odd;
}
```

Ex: 2 Algorithm for sum of the elements of an array

```
Algorithm array sum (a, n)
{
        s = 0; ———————— 1
        for i = 1 to n do ———— 1+n
        {
            s := s + a[i]; ———— n
        }
        return s; ———————— 1
}
                Time Complexity - 2n + 3
```

Ex:- 3 Algorithm for matrix addition

```
Algorithm for matrix Addition (a,b,c,m,n)
{
    for i = 1 to m do ———— m+1
    {
        for j = 1 to n do ———— m(n+1)
    {
        c[i,j] = a[i,j] + b[i,j] —— mn
    }
  }
}
```

Time Complexity = 2mn + 2m + 1

There are two approches to calculate time Complexity

1. frequency Count (or) step Count
2. Asymptotic notations

Components of time Complexity:

1. capacity of system: If speed of Computer is fast then output will be generated fastly other wise slowly.
2. Computer containing single processor or multiple processor

If Computer containing only single processor then output is Generated slowly.

If Computer containing multiple processor then output is Generated fastly.

Space Complexity :-

The amount of space that an Algorithm requires for its Execution is known as space Complexity.

There are three types of space Complexities

→ Best case space Complexity

→ Average case space Complexity

→ Worst case space Complexity

Components of space Complexity :-

① Instruction space :- space required in the computer in order to store instructions is known as Instruction space.

② Environmental stack :- Amount of space required to store partially Executed function

③ Data space :- Data space is required in order to store variables and Constants

Space Complexity of an Algorithm is Calculated by using two parts

① fixed part :- variables that have independent characteristics i.e Constants

② variable part :- variables that have dependent characteristics.

$$\boxed{S(P) = C + S_P}$$

space Complexity of program

Constant (or) fixed part

variable part (or) instance characteristics

Ex:-1

```
abc (x, y, z)
return x * y * z + (x-y)
```

$S(P) = C + S.P$

$= 3 + 0$

$= 3$

Ex:-2

```
int square (int a)
{
    return a*a
}
```

$S(P) = C + S.P = 1 + 0 = 1$

Ex:-3

```
sum (a, n)
{
    total = 0
    for i = 1 to n do
        total = total + a[i]
}
```

$S(P) = C + S.P = 3 + n$

Asymptotic Notations :-

By using Asymptotic Notations we can calculate time complexity of an algorithm. by using Asymptotic notations we can calculate Best case time complexity, average case time complexity, worst case time complexity of an Algorithm

There are 5 types of Asymptotic notations

① Big oh notation (O)

② Bigomega notation (Ω)

③ theta notation (θ)

④ little oh notation (o)

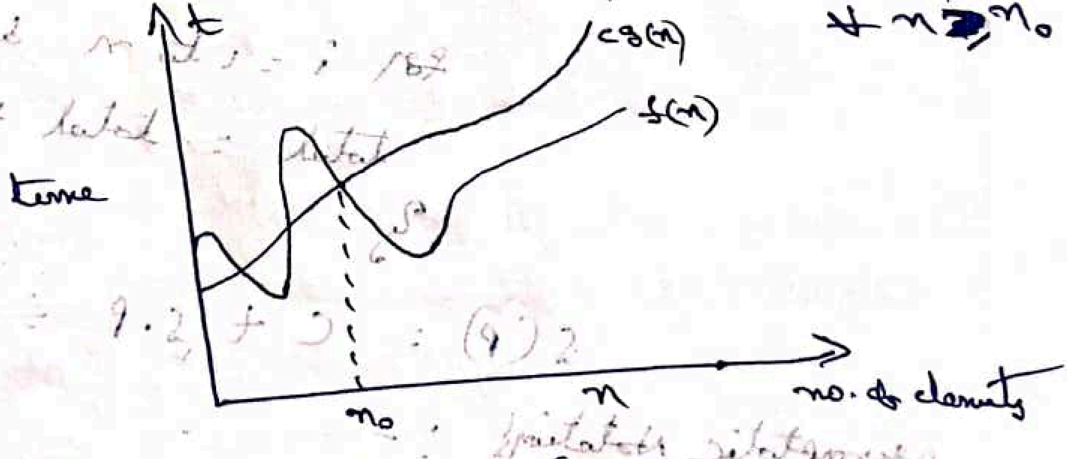
⑤ little omega notation (ω)

① Big oh notation :- (O)

-> It mainly represents upper bound of Algorithm run time

-> By using Big oh notation we can calculate maximum amount of time that an algorithm requires for its execution i.e worst case time complexity of an algorithm

Def :- Let $f(n)$, $g(n)$ be two non negative functions then $f(n) = O(g(n))$ iff there exist two positive constants c, n_0 such that $f(n) \leq c * g(n)$, $\forall n \geq n_0$



$f(n) = O(g(n))$

Ex:- $f(n) = 3n+2$, $g(n) = n$ prove that

$f(n) = O(g(n))$

Solu In order to prove $f(n) = O(g(n))$, we need to satisfy condition $f(n) \leq c * g(n)$ $\forall n \geq n_0$

$3n+2 \le c * n$

$3n+2 \le c * n$ [let $c=4$]

$3n+2 \le 4 * n$ [let $n \Rightarrow n_0 = 1$]

for $n_0 = 1 \Rightarrow 3(1)+2 \le 4 \times 1 \Rightarrow 5 \le 4$ false

for $n_0 = 2 \Rightarrow 3(2)+2 \le 4 * 2 \Rightarrow 8 \le 8$ true

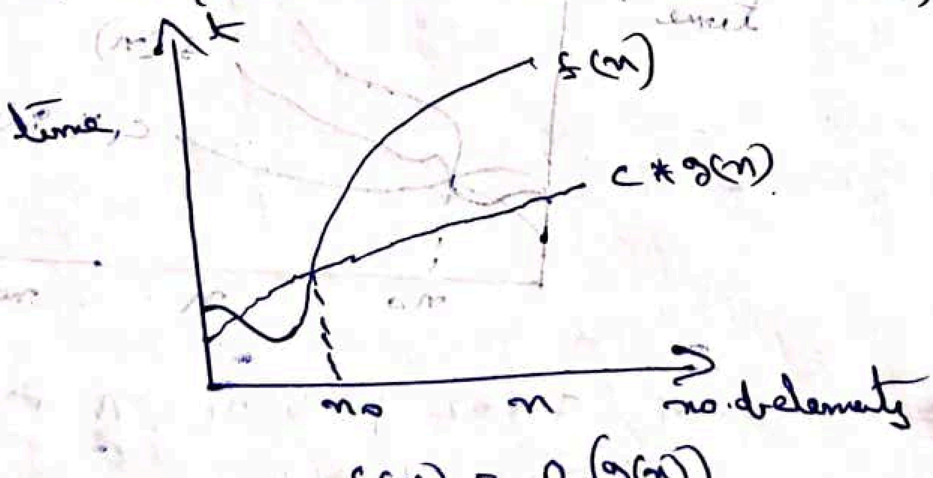
$\therefore f(n) \le c * g(n) \; \forall \; n \ge 2$

2) Big omega Notation :- (Ω)

→ It mainly represents lower bound of algorithm run time

→ By using Big omega notation, we can calculate minimum amount of time that an algorithm requires for its execution i.e Best case time complexity of an algorithm

Def :- let $f(n)$, $g(n)$ be two non negative functions then $f(n) = \Omega(g(n))$ iff there exist two positive constants c, n_0 such that $f(n) \ge c * g(n) \; \forall \; n \ge n_0$



$f(n) = \Omega(g(n))$

Ex :- $f(n) = 3n+2$; $g(n)$ prove that

$f(n) = \Omega(g(n))$

Sol: In order to prove $f(n) = \Omega(g(n))$, we need to satisfy condition $f(n) \ge c * g(n) \ \forall n \ge n$

$$3n+2 \ge c * n \ [\text{let } c=1]$$

$$3n+2 \ge 1 * n$$

for $n_0 = 1 \Rightarrow 3(1)+2 \ge 1 \cdot 1 \Rightarrow 5 \ge 1$ true

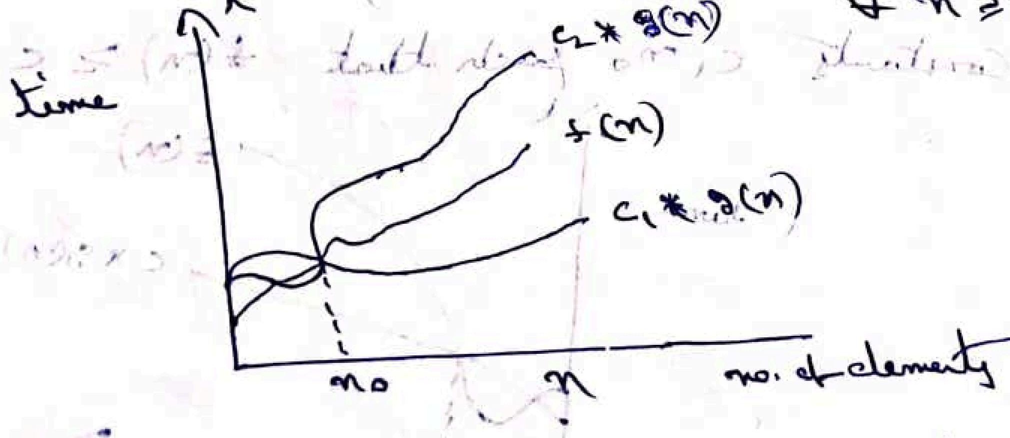
$$\therefore f(n) \ge c * g(n) \ \forall n \ge 1$$

③ Theta notation :- (θ)

→ It mainly represents average bound of Algorithm run time

→ By using theta notation we can calculate average amount of time that an algorithm requires for its execution i.e average case time complexity of an algorithm.

Def :- Let $f(n), g(n)$ be two non negative functions then $f(n) = \theta(g(n))$ if there exist three constants c_1, c_2, n_0 such that $c_1 * g(n) \le f(n) \le c_2 * g(n) \ \forall n \ge n_0$



Ex :- $f(n) = 3n+2$; $g(n) = n$ prove that $f(n) = \theta(g(n))$

Sol: In order to prove $f(n) = \theta(g(n))$, we need to satisfy condition $c_1 * g(n) \le f(n) \le c_2 * g(n) \ \forall n \ge n_0$

Let $c_1 = 1, c_2 = 4$

then $1 * n \leq 3n+2 \leq 4 * n$

for $n_0 = 1 \Rightarrow 1*1 \leq 3(1)+2 \leq 4*1$

$1 \leq 5 \leq 4$ false

for $n_0 = 2 \Rightarrow 1*2 \leq 3 \cdot 2+2 \leq 4*2$

$2 \leq 8 \leq 8$ true

$\therefore c_1 * g(n) \leq f(n) \leq c_2 * g(n) \; \forall \; n \geq 2$

④ Little oh notation :- (o)

Let f(n), g(n) be two non negative functions then $f(n) = o(g(n))$ such that

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$$

⑤ Little omega notation :- (ω)

Let f(n), g(n) be two non negative functions then $f(n) = \omega\, g(n)$ such that

$$\lim_{n \to \infty} \frac{g(n)}{f(n)} = 0$$

Order of Growth :- Measuring the performance of algorithm with i/p size n is order of Growth

n	$\log_2 n$	$n\log_2 n$	n^2	2^n
1	0	0	1	2
2	1	2	4	4
4	2	8	16	16
8	3	24	64	256
16	4	64	256	65,536
32	5	160	1024	4,294,967,296

from above it is clear that $\log_2 n$ is slowest growth function.

The 2^n (Exponential) function is fastest growth function.

practicle Complexities

Better ↑

$O(1) \longrightarrow$ Constant time

$O(\log n) \longrightarrow$ log time

$O(n) \longrightarrow$ linear time

$O(n \log n) \longrightarrow$ log linear time

$O(n^2) \longrightarrow$ Quadratic time

$O(n^3) \longrightarrow$ Cubic time

$O(n^2) \longrightarrow$ Polynomial time

$O(2^n) \longrightarrow$ Exponential time

$O(n!) \longrightarrow$ factorial time

↓ worst

Frequency Count (or) step Count method :-

In order to calculate time Complexity, we use frequency method. It specifies the number of times a statement to be executed.

Rules :- ① for comments, declarations step Count is 0

② for return and Assignment statements step count is 1

③ Ignore lower order exponents when higher order exponents are presents

④ Ignore Constants

Ex:- Algorithm for sum of elements in array

```
Algorithm Sum(A, n)
{
    s = 0 ———————————— 1
    for (i = 0; i < n; i++) ——— n+1
    {
        s = s + A[i]; ——— n
    }
    return s ———————— 1
}
```

A				
8	3	9	7	2
0	1	2	3	4

n = 5

i = 0

i = 1

i = 2

i = 3

i = 4

i = 5 ✗

$\therefore$ Time Complexity $= 1 + (n+1) + n + 1$

$= 2n + 3$

Hence Time Complexity of $O(n)$

Space Complexity :-

A	n
n	1
S	1
i	1

$\therefore\ S(n) = \underline{\overline{n+3}}$

Hence space Complexity of $O(n)$

Ex:- Algorithm sum of two matrices

```
Algorithm Add(A, B, n)
{
    for (i = 0; i < n; i++) ——— n+1
    {   for (j = 0; j < n, j++) — n(n+1)
{
    c[i,j] = A[i,j] + B[i,j] — n×n
  }
 }
}
```

$\therefore$ Time Complexity $= (n+1) + n(n+1) + n^2$

$= n+1 + n^2 + n + n^2$

$= 2n^2 + 2n + 1$

Hence time Complexity is $O(n^2)$

<u>Space Complexity</u> :-

A — n^2

B — n^2

C — n^2

n — 1

i — 1

j — 1

$S(n) = 3n^2 + 3$

Hence space Complexity is $O(n^2)$

<u>Ex</u> :- Algorithm for Matrix Multiplication

```
Algorithm Multiply (A,B,n)
{
    for (i=0; i<n; i++) — (n+1)
    {
        for (j=0; j<n; j++) — n(n+1)
        {
            c[i,j] = 0 ———— n·n
            for (k=0; k<n; k++) ———— n·n(n+1)
            {
n·n·n ————  c[i,j] = c[i,j] + A[i,k] * B[k,j];
```

}
}
}
} ∴ Time Complexity = $2n^3 + 3n^2 + 2n + 1$

Hence time Complexity as $O(n^3)$

Space Complexity :-

A —	n^2
B —	n^2
C —	n^2
n —	1
i —	1
j —	1
k —	1
	$3n^2 + 4$

Hence space Complexity as $O(n^2)$

pseudo code for Expressing Algorithms (or) Conventions

1. An algorithm is a procedure. It has two parts; the first part is head and the second part is body.

2. the Head section Consists of keyword Algorithm and name of the algorithm with parameter list

Ex:- Algorithm name (P1, P2, P3, ..., Pn)

the Head section also has the following:

```
// problem Description:
// input:
// out put:
```

3. In the body of an algorithm various programming constructs like if, for, while and some statements like assignments are used.

4. the Compound statements may be enclosed with

{ and } brackets. if, for, while can be open and closed by {, } respectively. proper indention is must for block.

5. Comments are written using // at the begining

6. The Identifier should begin by a letter and not by digit. It contains alpha numeric letters after first letter. No need to mention data types.

7. The left arrow ":=" used as assignment operator. Eg:- V:=10

8. Boolean operators (True, false), logical operators (AND, OR, NOT) and relational operators (<, <=, >, >=, =, ≠, <>) are also used.

9. input and output can be done using read and write

10. Array [ ], if then else condition, branch and loop can be also used in algorithm

Ex:- Algorithm for fibonacci Sequence using non-recursive

```
fibo(n)                                    n=0|1      n>1
{
   if (n<=1) then ----------------------   1 -------- 1
   write(n); ---------------------------   1
   else
   {
      a=0; b=1 ------------------------------------- 1
      for i=2 to n do ------------------------------ n
      {                                             n-1
           c=a+b ---------------------------------- n-1
       a=b  ~~b=a~~ ------------------------------- n-1
       b=c  ~~a=c~~ ------------------------------- 1
      }  write(c); --------------------   2         4n
```

Recursive Algorithm :-

A recursive function is a function that is defined in terms of it self. Similarly, an algorithm is said to be recursive if the same algorithm is invoked in the body.

Ex :- Towers of Hanoi

```
Algorithm TOH(n, source, destination, inter)
    {
        if (n>0) then
        {
            TOH (n-1, source, inter, destination)
            write (" Move a Disk from(tower) source to
                                 destination");
            TOH(n-1, Inter, destination, source)
        }
    }
```

Recurrence relation of Recursive algorithms

A Recurrence relation is an equation that recursively defines a sequence. the recurrency can be solved by " Substitution method".

```
        Int factorial (int n)
    {
            If (n = = 1)     }
            {                }  Base
                return 1;    }
            }
            else
            {
A               return n * factorial(n-1); } recurrence
            }
    }
```

$$T(n) = \begin{cases} T(n-1)+2 & \text{if } n>1 \\ 1 & \text{if } n=1 \end{cases}$$

so we have $T(n) = T(n-1)+1$ — (1)

$\Rightarrow T(n-1) = T(n-2)+1$ — (2)

$T(n-2) = T(n-3)+1$ — (3)

now (1) $\Rightarrow T(n) = T(n-2)+1+1$

$T(n) = T(n-3)+1+1+1$

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

$\vdots$

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

If $n-k=1$ then Algorithm will stop

$\Rightarrow k = n-1$

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

$= n-1+T(\cancel{n}-\cancel{n}+1)$

$= n-1+T(1)$

$= n-1+1$

$= n$

Hence the efficiency for the recursive function is "n".

(2) Solve the recurrence relation and find the time Complexity $T(n) = 2T(n/2)+n$ and $T(1)=2$

Sol: we have $T(n) = 2T(n/2)+n$; $T(1)=2$ — (1)

from (1) we have $T(n/2) = 2(n/4)+n/2$ — (2)

$T(n/4) = 2(n/8)+n/4$ — (3)

from (2) then (1) $\Rightarrow T(n) = 4T(n/4)+2n$ — (4)

From (3) then (4) $\Rightarrow$ $T(n) = 8\,T(n/8) + 3n$

$$\Rightarrow T(n) = 2^3\, T\left(\frac{n}{2^3}\right) + 3n$$

So general equation may be

$$T(n) = 2^k\, T\left(\frac{n}{2^k}\right) + kn$$

Let, $\frac{n}{2^k} = 1 \Rightarrow n = 2^k$

$$\Rightarrow \log_2 n = k$$

So, new general equation becomes

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

$$T(n) = n \cdot 2 + n\log_2 n$$

$$T(n) = 2n + n\log_2 n$$

$\therefore$ Time Complexity $O(n\log_2 n)$

3) Solve the recurrence relation $T(n) = T(n-1) + n$ and $T(1) = 1$. Also find the Time Complexity.

Soln we have $T(n) = T(n-1) + n$ —— (1) and $T(1) = 1$

replace n by $n-1$ in eqn (1)

$$\Rightarrow T(n-1) = T(n-2) + (n-1) \text{ — } (2)$$

Substitute eqn (2) in eqn (1), we get

$$T(n) = T(n-2) + (n-1) + n \text{ —— } (3)$$

Again replace n by $n-2$ in eqn (1),

$$\Rightarrow T(n-2) = T(n-3) + n-2 \text{ —— } (4)$$

putting eqn (4) in eqn (3), we get

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

the General equation is

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

If $n-(k+1) = 1$ then algorithm will stop

$$n-k-1 = 1$$

$$n-k = 2$$

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

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

$$= 1 + 2 + \cdots + n$$

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

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

∴ Time Complexity $= O(n^2)$

Contrast the Algorithm and Psuedocode :-

Algorithm	Psuedocode
① It is a step by step description of the solution	① it is an easy way of writing algorithms for users to understand.
② It is always a real algorithm and not fake codes.	② there are fake codes
③ they are a sequence of solutions to a problem.	③ they are representations of algorithms
④ It is a systematically written code.	④ these are simpler ways of writing codes.
⑤ They are an unambiguous way of writing codes.	⑤ They are a method of describing codes written in an algorithms

(6) They can be Considered pseudocode.	(6) They can not be Considered algorithms
(7) there are no rules to writing algorithms	(7) Certain rules to writing pseudocode are there.

Q Differences between algorithm and program :-

Algorithm	Program
(1) It is a step by step process for solving Computational problems.	(1) A program is nothing but set of instructions (or) exacutable code.
(2) An Algorithm is designed by domain knowledge (designer).	(2) The program can be implemented by specific programmer.
(3) An algorithm is done at design time.	(3) program is done at implementation time.
(4) Any language Ex:- Mathematical notation, General english statements.	(4) Any programming languages Ex:- C, C++, Java, .Net
(5) After Completion of your algorithm you have to analyze based on some criterias, such as time Complexity and space Complexity	(5) After Completion of writing your program you have to do test your program.

*

Q Differences and Comparison b/w " Big oh, omega and theta notations.

Big oh	omega	Theta
① the function $f(n) = O(g(n))$ iff their exist constant c and +ve integer n_0 such that $f(n) \leq c \cdot g(n) \; \forall \; n \geq n_0$	① the function $f(n) = \Omega(g(n))$ iff their exist constant c and +ve integer n_0 such that $f(n) \geq c \cdot g(n) \; \forall \; n \geq n_0$	① the function $f(n) = O(g(n))$ iff their exist constants c_1, c_2 and +ve integer n_0 such that $c_1 g(n) \leq f(n) \leq c_2 \cdot g(n) \; \forall \; n \geq n_0$
② It is indicated by " O "	② It is indicated by " Ω "	② It is indicated by " Θ "
③ Represents worst case time complexity	③ Represents Best case time complexity	③ Represents average case time complexity
④ It takes maximum amount of time	④ It takes minimum amount of time.	④ It takes average amount of time.
⑤ It is an upper bound	⑤ It is an lower bound	⑤ It is an average bound.

Q **Amortized Analysis :-**

Def :- Finding average running time per operation over the worst case sequence of operations

* Amortized analysis does not allow random numbers

* Amortized analysis and average case analysis are different.

* In case of average analysis we can find averaging all the inputs, but where as amortized analysis we are averaging over a sequence of operations.

* To compute amortized analysys we can follow the following Techniques.

① Aggregate method

② Accounting method

③ Potential method.

Aggregate method :-

* Aggregate method is used to find the total cost per "n" no. of worst case sequence of operations.

* Aggregate method is equal to $\frac{T(n)}{n}$

where $T(n)$ = Total cost

n = no. of operations

Accounting method :-

* By using accounting method we can find individual cost per an operation.

* finally we found total cost i.e $\frac{T(n)}{n}$.

Potential method :-

* Potential method is like a accounting method but operations depend upon its lower cases and upper cases.

Note :- upper bound, lower bound and average bound can be represented by the following statement.

$$1 < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 < \ldots\ldots < 2^n < 3^n < \ldots < n^n$$

Lower bound — Average bound — upper bound.

1 Explain about analysis Frame work briefly?

It is a systematic approach applied for analyzing any given algorithm

1. Measuring Time Complexity
2. Measuring space Complexity
3. Computing order of Growth of algorithms
4. Measuring input size
5. Measuring Running time.
6. Computing Best case, worst and average case efficiencies

(1) <u>Measuring Time Complexity</u> :- Def. with example

(2) <u>Measuring Space Complexity</u> :- Def, formula, example.

(3) <u>Computing order of Growth</u> :- i/p size n.

(4) <u>Measuring input size</u> :-

-> the efficiency of an algorithm can be computed as a function.

-> Input size is passed as a parameter

-> It can be (i) Exact value
(ii) Approximate value

Ex:- Spell — checking Algorithm
-> Number of characters }
-> Number of words } — input size.

(5) <u>Measuring Running time</u> :-

From an algorithm
-> Identify basic operation.
-> understand the concept of basic operation.
-> Compute total number of time taken by basic operation.

Formula used is $T(n) = C_{op} \cdot C(n)$

where

$T(n)$ — Running time of basic operation

C_{op} — Time taken by basic operation to execute

$C(n)$ — Number of times the operation needs to be executed.

(6) Computing Best, average and worst case efficiencies :-

Linear search :-

→ If the searching element is at starting index is the Best case. Time Complexity is $O(1)$

→ If the searching element is at last index (or) not present in array, is the worst case. Time Complexity is $O(n)$

→ average time Complexity $= \dfrac{1+2+3+\dots+n}{n}$

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

$$= \frac{n+1}{2}$$

∴ Time Complexity is $O(n)$.