DAA – Design and Analysis of Algorithms

Algorithm:- Algorithm is a finite set of instructions to complete the problem (or) to solve the problem of an algorithm must satisfy the following criteria:-

1. Input
2. output
3. Definiteness
4. finiteness
5. Effectiveness

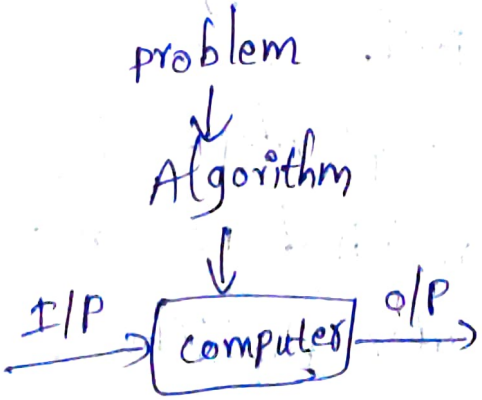
Input: Zero (or) More quantities are supplied externally.

output:- Atleast one quantity is produced.

definiteness: Each quantity is clear & unambiguous

finiteness: The algorithm must be terminated with finite no.of steps.

effectiveness:- Every instruction must be basic enough and must be feasible.



Algorithm for problem solving:-

The main steps for problem solving are:

1. problem definition
2. algorithm design / specification
3. algorithm analysis
4. Implementation
5. Testing
6. Maintainance

① problem definition:- what is the task to be accomplished.

ex:- addition of two numbers, factorial

② algorithm design:- blue print (or) action of plan

ex:- diagrams, pseudo code, er diagrams

③ algorithm analysis:-
1. space complexity
2. Time complexity
3. computer algorithm

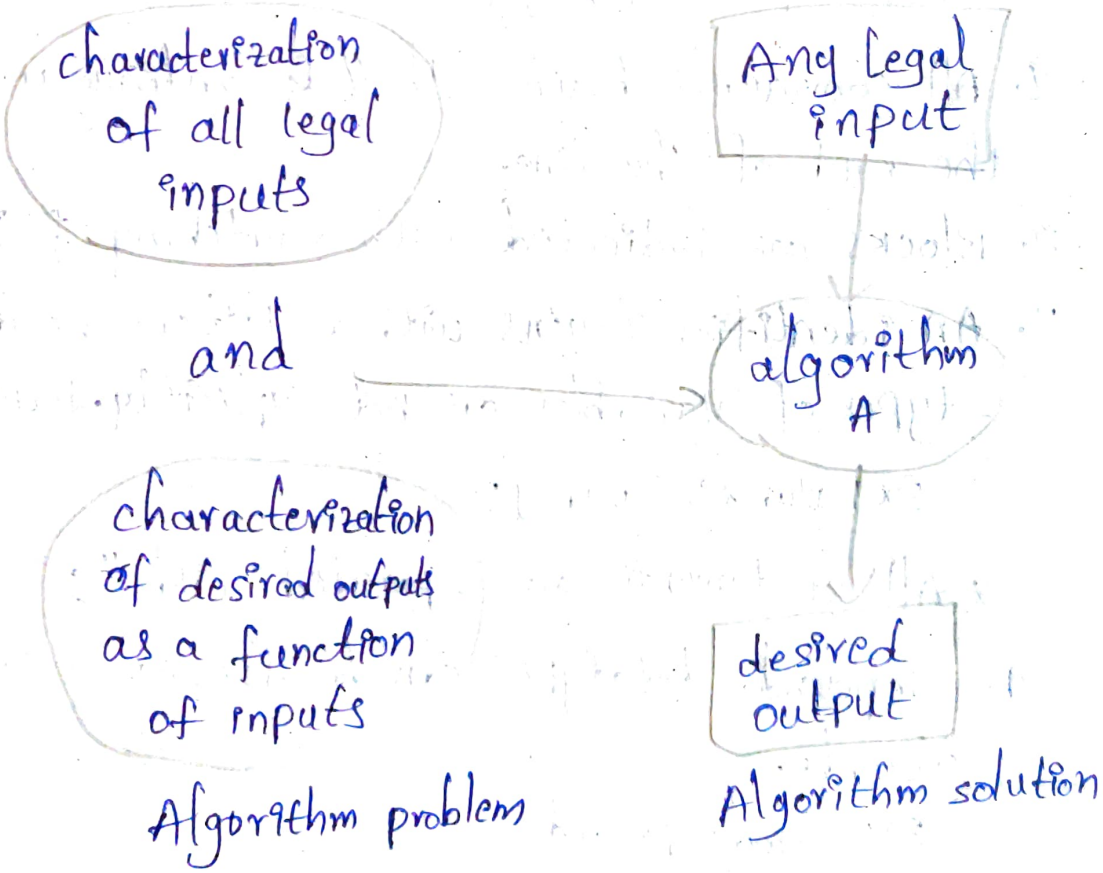
1. space complexity — how much space is required.
2. Time complexity — how much run time is required to run the algorithm.
3. computer algorithm — step by step process to solve the given problem.

* computational complexity and efficient implementation of an algorithm is dependence upon the data structures.

④ Implementation:- Is nothing but coding.
ex:- by various languages C, C++, Java, ...

⑤ Testing:- Integrate the feedback from the customers, fix bugs, ensures compatability accross different versions.

⑥ maintainance:- release updates and fix bugs.



Pseudo code for expressing algorithms:-

Algorithm specification:

Algorithm can be specified in 3 ways

1. Natural Language
2. Graphical Representation
   eg: UML, ER diagrams, DMD
3. Pseudo code method

1. Natural Language:-

e.g:- English

3. Pseudo code:-

In this method algorithms are specified using programming language.

Ex:- Pascal, Algol.

1. All comments begin with // & continue until the end of the line.
2. Blocks are indicated with matching { }
3. An identifier begins with a letter. The data-types of variables are not explicitly declared.

   Ex: var x; var y;

   * All statements are separated with ";"
4. compound datatypes can be formed with records.

   eg:- Node Record
   {
   datatype-1 data-1;
   :

```
datatype-n  data-n;
node *link;
}
```

5. Assignment operator [:= (or) ←]

Assignment of values to the variables. This operator can be represented as shown.

6. There are 2 boolean values

TRUE (or) FALSE

logical operations: AND, OR, NOT

RELATIONAL operators: [<, ≤, = ≥, >]

7. looping statements:

Here the following looping statements are employed: for, while, repeat until

while loop:

```
while <condition> do
{
    <statement 1>
      :
    <statement 2>
}
```

for loop:-

```
for variable := value-1 to value-2 step step do
{   <statement 1>
      :
    <statement 2>
```

Repeat until:

```
repeat
  <statement 1>
    :
  <statement n>
  until <condition>
```

8* conditional statements:-

i) if ii) if else iii) case

If

```
if <condition> then <statement>
```

If else

```
if <condition> then <statement-1>
  else <statement-2>
```

Case

```
case
{
  <condition 1>: <statement-1>
    :
  <condition n>: <statement-n>
  else
    <statement n+1>
}
```

9) Input & output are done using read & write instructions.

10) There is only 1 type of procedure

Algorithm <Name> (<Parameter lists>)

```
0) Algorithm Max(A,n)
1) //A is an array of size n
2) {
3) Result := A[1];
4) for I:=2 to n do
5)     if A[I] > Result then
6)     Result := A[I];
7)     return Result;
8) }
```

performance Analysis:-

The space complexity of a program is the amount of main memory, it needs to run the completion.

→ The space needed by the algorithm consists of following components.

(i) The fixed static part independent of characterstics <numbersize> of input & output

This part typically includes instruction space(code space), space for simple variable & fixed size component variable, space for constants etc.

(ii) A variable dynamic part. It consists of space needed by the component variables whose size depends on particular problem

instance at runtime been solved, the space needed by Reffered variables & the recursion stack space.

$S(P) = C + SP$ — C is constant

fixed + variable

```
Algorithm ADD(x,n)
{
  Total = 0.0;        ——— 1
  for i=1 to n do     ——— n
  Total = Total + x[i]; — 1
  Return Total;       ——— 1
}
```

n+3 → n

O(n)

space complexity

```
Algorithm xyz(x, y, z)
{
  return (x+y+y*z + (x+y-z)/(x+y)+40;
}
```

→ In the above algorithm there are no instance characterstics & the space needed by x, y, z is independent of instant characterstics. Therefor we can write $S(x, y, z) = 3+0$

so O(1).

*
```
Algorithm Rsum(x, n)
{
  if (n ≤ 0) then return 0.0;
  else return (Rsum (x, n-1) + a[n]);
}
```

TIME complexity:-

how much amount time is required to run the program.

sequential statements:-

```
statement 1;
statement 2;
---------
statement N;
```

* If we calculate the time complexity it would be like this

total = time(statement 1) + time(statement 2) +
------ + time (statement N)

Let's use T(N) is total time in a function of N is time complexity taken by statement or group of statement.

T(n) = t(statement 1) + t(statement 2) + ----
+ t(statement N)

eg:-

```
function square sum(a,b,c){
    const sa=a*a;  -> O(1)
    const sb= b*b;  -> O(1)
    const sc=c*c;  -> O(1)
    const sum= sa+sb+sc;  -> O(1)
    return sum;  -> O(1)
                    O(5)
```

∴ O(5) is equal to O(1)

* In the above example each stmt is basic operation (math & assignment operation). Each stmt required o(1). If we add all stmts time will be o(1). It does not matter numbers are 0, 999, 199--- it will perform n no. of operations. Ex:- o(999) = o(1)

Conditional statement:-

```
if (is valid) {
    statement 1;
    Statement 2;
} else {
    statement 3;
}
```

T.C = o(n)

```
if (is valid) {
    array sort();
    return true;
} else {
    return false;
}
```

T.C = o(n)

Ex:- n = 8 then 8 x 3 + 8 => 24 + 8 → nlogn

* conditional stmt time complexity is o(n).

$O(n \log n) + O(n)$

* so, $(n \log n)$ has the higer order than 'n',
* so, the time complexity is expressed as $O(n \log n)$.

linear time loops:-

Ex:-
```
for(let i=0; i<array.length; i++){
    statement 1;   → O(n)
    statement 2;   → O(n)
}
```

Time complexity → $O(n)$

Constant time loops:-

Ex:-
```
for(let i=0; i<4; i++){
    statement 1;
    statement 2;
}
```

∴ $O(1)$ is time complexity

logarithamic time loops:-

Ex:-
```
function fn1(array, target, low=0, high=
        array.length-1) {
  let mid;
  while(low<=high){
  mid = (low+high)/2;
  if(target<array[mid])
      high = mid-1;
```

```
    else if (target > array[mid])
        low = mid + 1;
    else
        break;
  }
  return mid;
}
```

∴ binary search → time complexity is ⇒ o(logn)

* This function devides the array by its middle point on each iteration. The while loop will execute the amount of times that we can divide the array.length in half. we can calculate this using log() function.

neasted Loop statements :-

Ex:-

```
for(let i=0; i<n; i++){
    statement 1;
    for(let j=0; j<m; j++){
        statement 2;
        statement 3;
    }
}
```

∴ time complexity is o(nm)

function call statements:-

Ex:-

```
for (let i=0; i<n; i++)
{
    fn1();
    for(let j=0; j<n; j++)
    {
        fn2();
        for(let k=0; k<n; k++)
        {
            fn3();
        }
    }
}
```

∴ time complexity is — $o(n^3)$ [∵ If three functions are constants.]

∴ time complexity is — $o(n^2)$ [when two functions are constants and third one is some addition function]

Time complexity methods:-

Time complexity is measured with the help of following factors:

1. Inputs
2. quality of the code generated by the compiler used to create the object program
3. nature and speed of the instructions on the machine used to execute the program

④ The time complexity of the program is underlying the program.

* Time complexity can be measured in two ways:

① step count method

② tabular method (frequency count)

① Tabular method:-

Ex:- array sum

statement	steps/execution	*	frequency	total
1. Algorithm sum(a,n)	—		—	—
2. {	—		—	—
3. s=0.0;	1	*	1	1
4. for I=1 to n do	1	*	n+1	n+1
5. s=s+a[I];	1	*	n	n
6. Return s;	1	*	1	1
7. }	—		—	—
Total:-				2n+3 ⇒ O(n)

∴ so, the total time complexity is O(n).

Ex:- matrix addition

statement	steps/execution	frequency	total
1. Algorithm add(a,b,c,m,n)	—	—	—
2. {	—	—	—
3. for i:=1 to m do	1	* m+1	m+1
4. for j:=1 to n do	1	* m*(n+1)	mn+m
5. c[i][j]:=a[i][j] +b[i][j]	1	* m*n	mn
6. }	—	—	—

∴ so, the time complexity is

⇒ 2mn+2m+1.

Ex:-

statement	steps/execution	frequency	total
1. Algorithm add(a,b,c,m,n)	—	—	
2. {	—	—	
3. for i:=1 to m do	1	m+1	
4. for j:=1 to m do	1	m(m+1)	
5. {	—	—	
6. c[i][j]=0;	1	m	
7. for i:=k to m do	1		
8. c[i][j]:=c[i][j]	1		
9. } +a[i][k]*b[k][j];	—		

② step count method:

① Ex:-1.

```
1. Algorithm sum(a,n)        → 0
2. {                         → 0
3.   s=0.0;                  → 1
4.   for I=1 to n do         → n+1
5.     s=s+a[I];             → n
6.     Returns;              → 1
7. }                         2n+3
```

Ex:-2 (reverse of a given number)

```
1. Algorithm rev(n)          → 0          —
   {                         → 0          —
     rev=0                   → 1          1
     while(n>0)              → n+1        n+1
     {                       → 0          —
       r=n%10;               → n          n
       rev=rev*10+r;         → n          n
       n=n/10;               → n          n
     }                       → 0          —
   }                         → 0          —
                             4n+2         4n+2
```

performance analysis:-

asymptotic notations:-

* asymptotic notations are used in performance analysis and used to characterize the complexity of an algorithm.

① big(O) notation:-

The function $f(n)$ is $g(n)$ if and only iff their exists two positive constants c, no

$f(n) = \boxed{f(n) \leq c.g(n)} \quad \forall n \geq no$

ex:- $f(n) \leq c.g(n)$

① $3n+5 \leq 4n$

$\boxed{n=1}$ $8 \leq 4$ ✗

$\boxed{n=2}$ $11 \leq 8$ ✗

$\boxed{n=3}$ $14 \leq 12$ ✗

$\boxed{n=4}$ $17 \leq 16$ ✗

$\boxed{n=5}$ $20 \leq 20$ ✓

$\boxed{n=6}$ $23 \leq 24$ ✓

② $f(n) = 3n+2$ then prove than $f(n) = O(n)$?

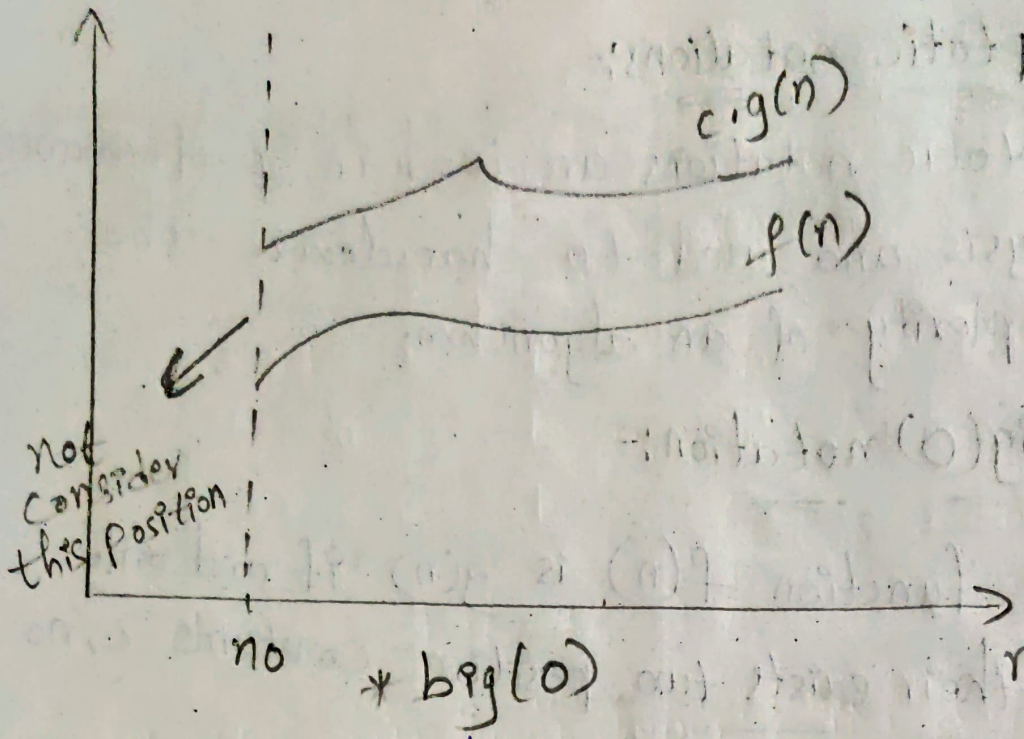
sol:- $\boxed{n=1}$ $5 \leq 1$ ✗

$\boxed{n=2}$ $8 \leq 2$ ✗

$\boxed{n=3}$ $11 \leq 3$ ✗

$\boxed{n=4}$ $14 \leq 4$ ✗

$\boxed{n=5}$ $= 15+2 \Rightarrow 17 \leq 5$



* big (O) notation

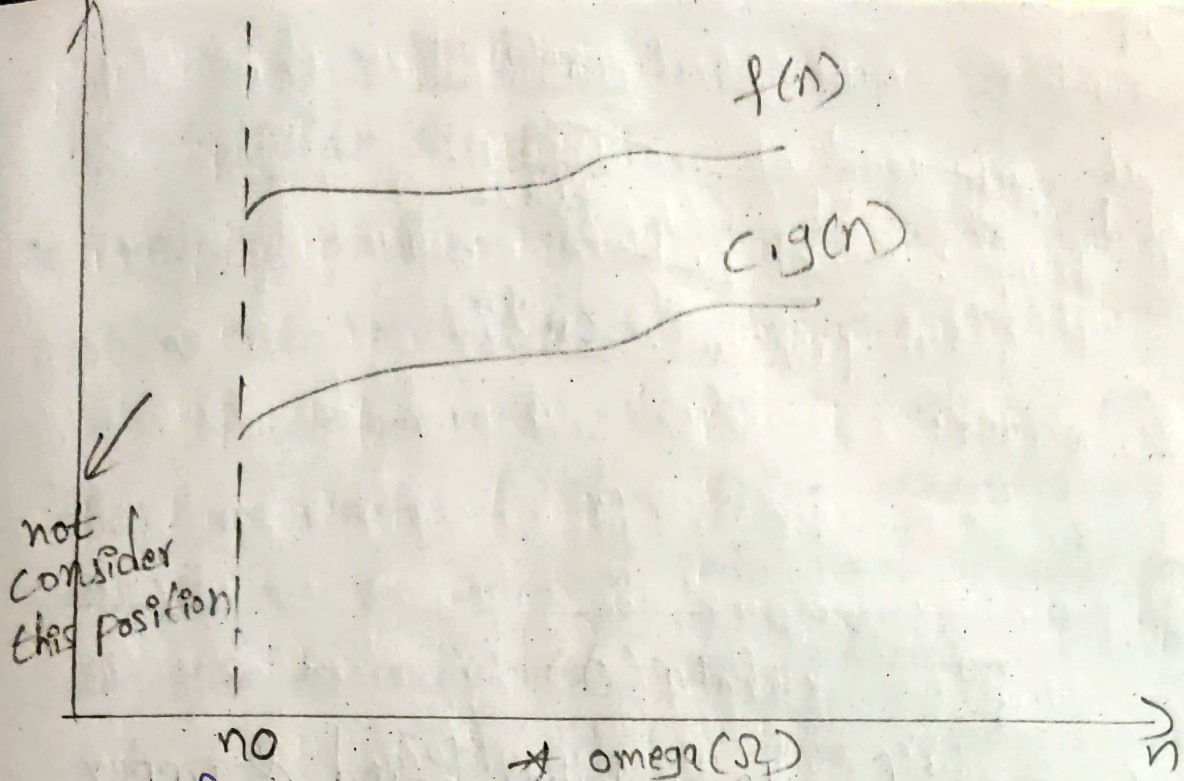
* big (O) notation gives an upper bound and a function.

* The upper bound of function $f(n)$ indicates that the function will be the worest case. It does not consume the more than the computing power.

② omega (Ω) notation:-

The function $f(n)$ is $g(n)$ if and only iff their exists two positive constants C, no.

$$f(n) \geq c.g(n) \quad \forall \; n \geq n_0$$



* The lower bound of function f(n) indicate that the function will be best case. It does consume the more than the computing power.

ex:-

① $f(n) = 3n+2$ $(f(n) = \Omega(n))$ $\forall n \geq 1$

$3n+2 \geq c(n)$

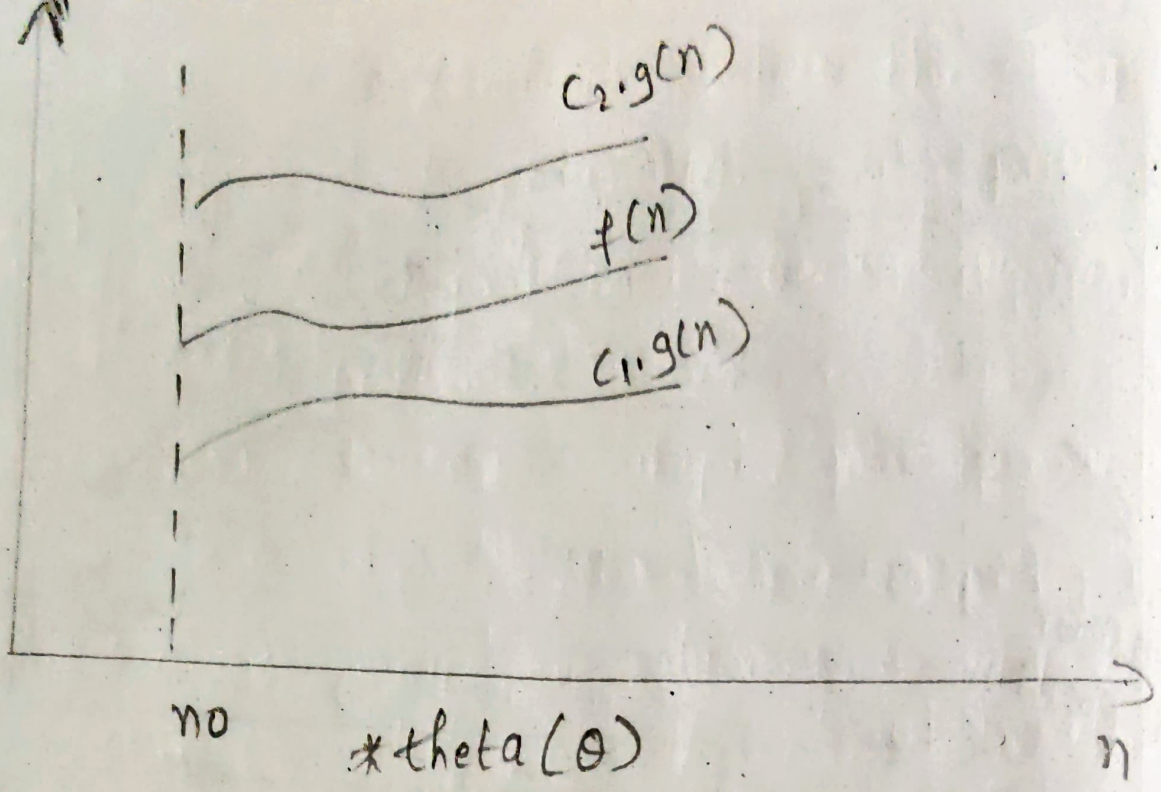
$n=1$ $\quad 5 \geq 1$ ✓

$n=2$ $\quad 8 \geq 2$ ✓

③ theta (Θ) notation:-

* $f(n) = \Theta g(n)$ if their exists three positive constants c_1, c_2, n_0.

$$c_1|g(n)| \leq f(n) \leq c_2|g(n)|$$



* for some functions lower bound & upper bound will be same. big(O) & omeg(Ω) will have the same function. for example:-
~~solve~~ sum of 'n' numbers, Mixmum & minimum number in the array. The computing time is $\boxed{f(n) = \Theta(n)}$