11/3/2023

UNIT–I

→ An algorithm is a sequence of steps to solve a problem.

→ Design and analysis of algorithm is very important for designing algorithm to solve different types of problems in the branch of computer science and information technology.

→ What is design algorithm?

→ Algorithm design refers to a method or a mathematical process for problem-solving and engineering algorithms.

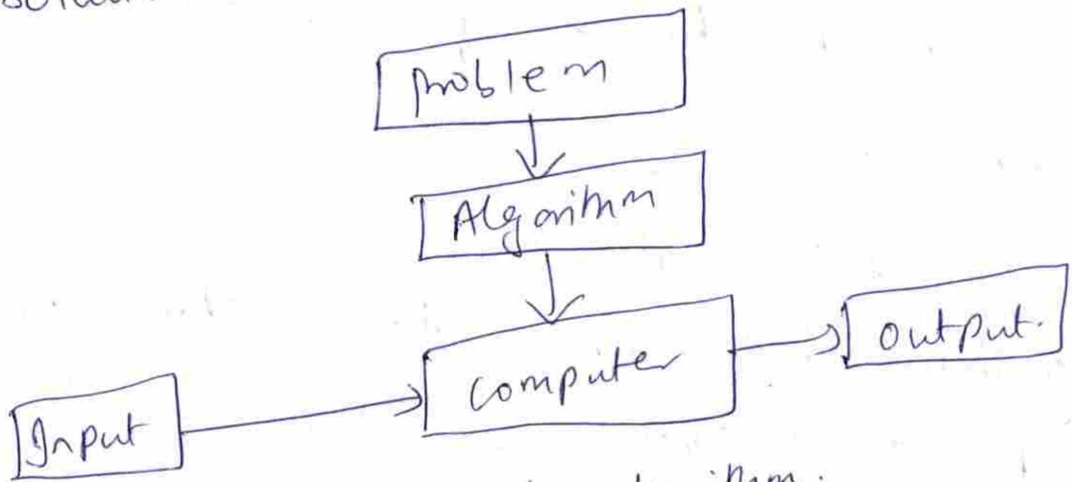
→ The design of algorithms is part of many solutions theories, such as Divid & Conquer or dynamic programming within operation research.

→ What is design analysis?

Design analysis is essentially a decision-making process in which analytical tools derived from basic sciences, mathematics, statistics and Engineering fundamentals are utilized to develop a product model that can be converted into an actual product.

Algorithm:- An algorithm is typically refers to a set of instructions that can be executed by a computer to produce the desired result.

→ An algorithm is not a solution to a problem, it ~~is~~ defines the procedure for getting solution to a problem.



notation of algorithm.

Properties of algorithm

5 properties of an algorithm

① **Input:-** An algorithm has zero or more "input" quantities that are given to it initially before the algorithm begins or dynamically as the algorithm runs.

Input refers to data (facts, figures, numbers) provided to the algorithm on which the computation is performed.

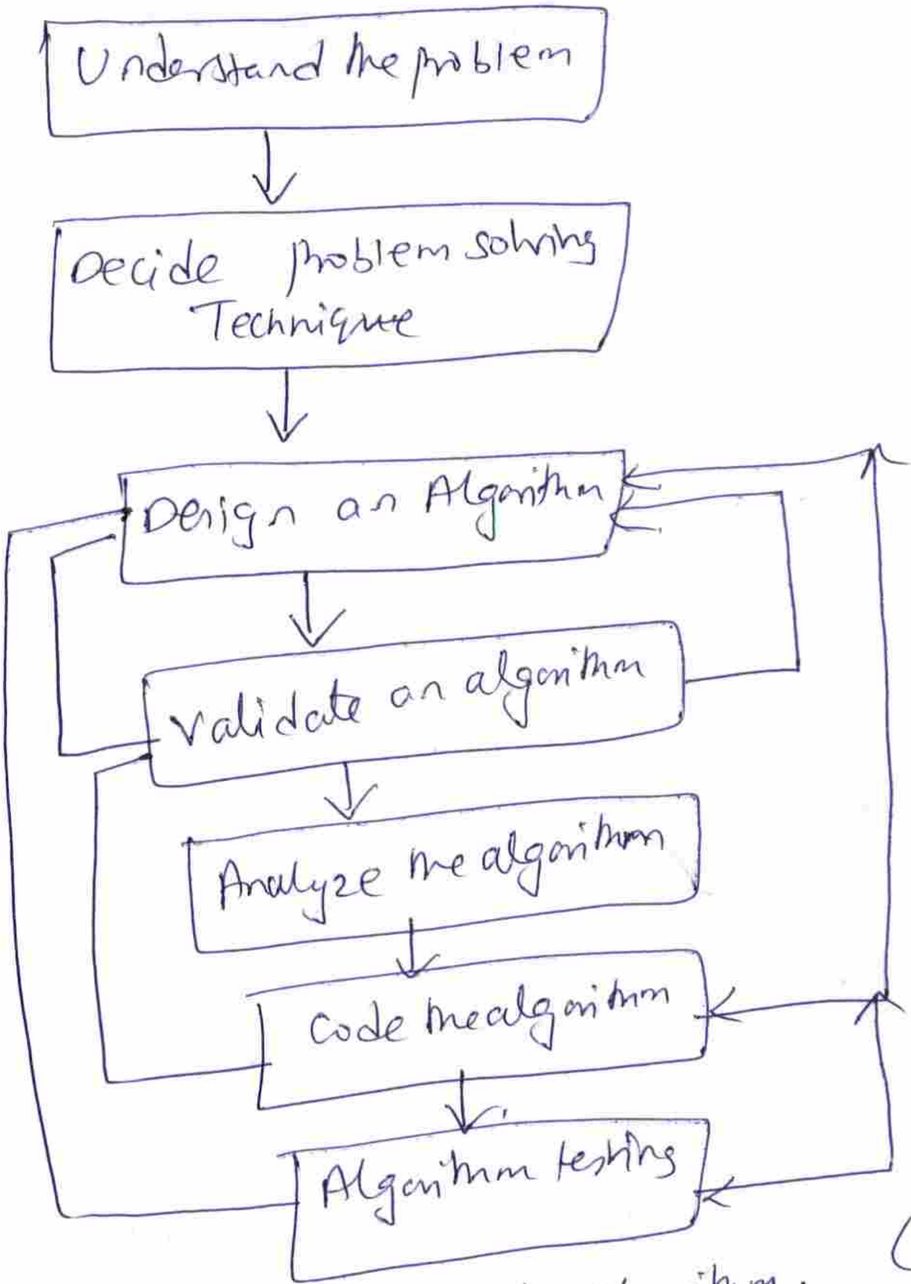
Output:- An algorithm has one or more outputs that ③
have a specified relation to the inputs.

(3) Definiteness:- Each step of algorithm must be precisely defined. Each action to be carried out must be rigorously and unambiguously specified for each case.

(4) Finiteness:- An algorithm must always terminate after a finite no. of steps, each of which may require one or more operations.

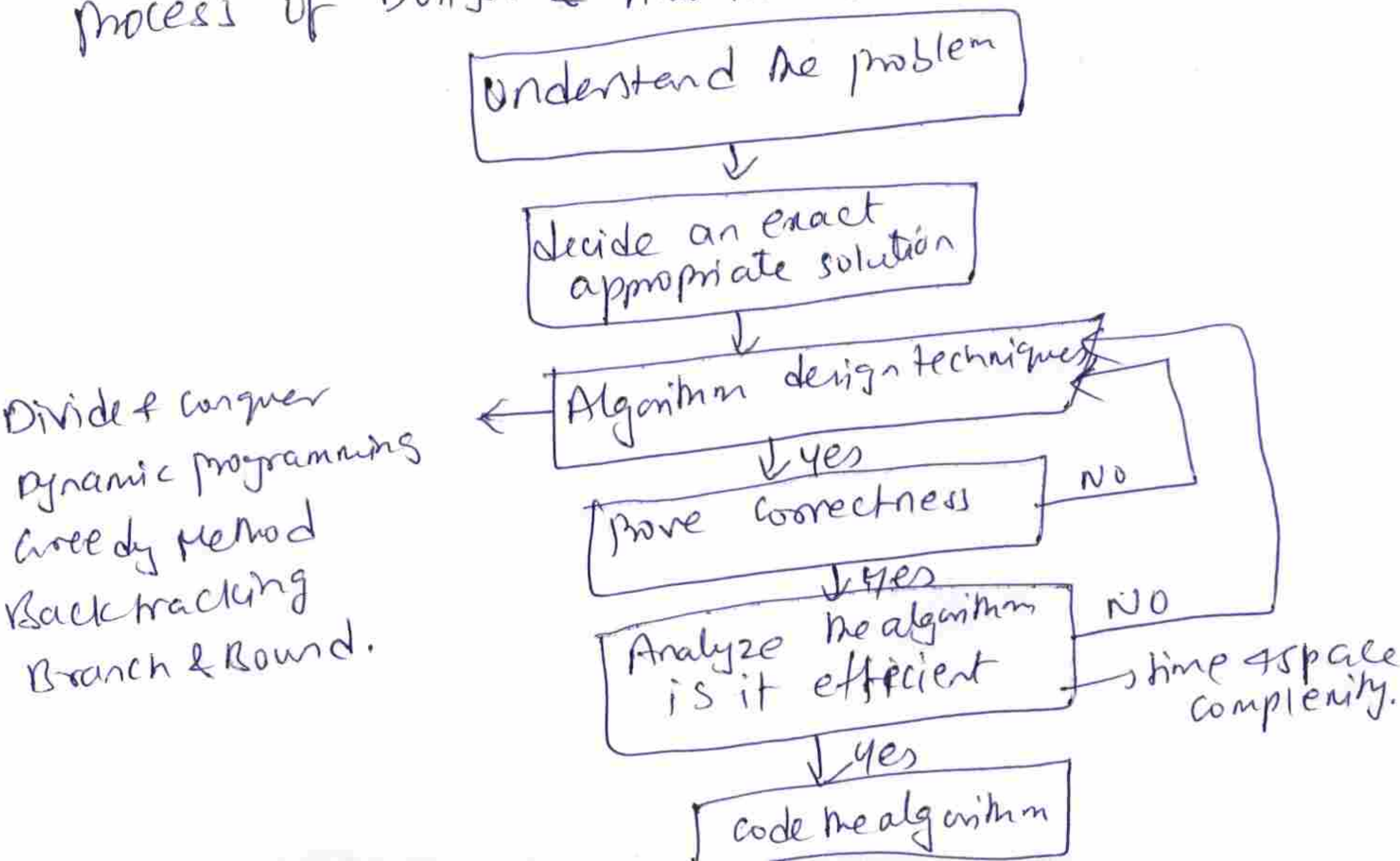
succen results → better results
(5) Effectiveness:- An algorithm is also generally expected to be effective, in the sense that its operations must all be sufficiently basic that they can in principle be done exactly and in a finite length of time by someone using pencil and paper.

→ i/e output must be feasible and logical according to the provided input and resources.



(or)

Process of Design & Analysis of Algorithm.



The 4 distinct areas of studying algorithm are

(1) How to devise algorithms - creating an algorithm.

i.e It is an art which never fully automated.

→ A major goal is to study various design techniques that have proven to be useful.

→ By mastering these design techniques/strategies it will become easier for you to device new and useful algorithms.

→ Some of techniques may already be familiar, and some have been found to be useful.

→ Dynamic programming is a technique which is useful in the fields other than computer science.

(2) How to validate algorithms - Definiteness.

→ Once the algorithm is devised, it is necessary to show that it computes the correct answer for all possible legal inputs.

→ i.e Algorithm validation. The algorithm need not as yet be expressed as a program.

→ The purpose of validation is to ensure is that this algorithm will work correctly independently

→ It is refered to as program verification.

(3) How to analyze algorithm:- As an algorithm is executed, it uses the computer's central processing unit (CPU) to perform operations and its memory to hold the program and data.

→ Analysis of algorithms or performance analysis refers to the task of determining how much computing time and storage replace.

→ Analyze the algorithm based on time and space complexity.

→ The amount of time needed to run the algorithm is called time complexity.

→ The amount of memory needed to run the algorithm is called space complexity.

(4) How to test program:-

→ Testing a program consists of two phases
(1) Debugging (2) profiling.

Debugging:- It is the process of executing program on sample data sets to determine whether faulty results occurs, if so correct them.

→ Debugging can only point to the "presence of errors but not the absence."

Profiling:- profiling or performance measurement is the process of executing a correct program on data sets and measuring the time and space it takes to compute the results.

The process of finding bugs or errors in a software product is termed testing. which is done manually by a tester or can be automated. Debugging is the process of resolving the bugs found in the testing phase. Developers and programmers are in charge of debugging and it can't be automated.

Algorithm specifications-

Algorithm can be described in 3 ways.

(1) Natural language like English:-
When this way is choosed care should be taken, we should ensure that each & every statement is definite.

(2) Graphic representation called flowcharts
This method will work well when the algorithm is small & simple.

(3) Pseudo-code method:-
This method describe algorithm as program, which resembles language like pascal & algol.

Algorithm specification:-

Pseudo-code conventions for expressing algorithms:-

(1) Comments begin with // and continue until the end of line.

(2) Blocks are indicated with matching braces { and }. A compound statements i.e collection of simple statements can be represented as a block.

Statements are delimited by ;

(3) An identifier begins with a letter. The datatype of variables are not explicitly declared.

(4) Compound data types can be formed with records.

```
Node.Record
{
datatype → data_1;
:
datatype_n data_n;
node *link;
}
```

→ link is a pointer to the record type node.
→ Individual data items of a record can be accessed with → and period (.)

(5) Assignment of values to variables is done using the assignment statement.

<variable> := <expression>;

(6) There are two boolean values TRUE and FALSE
→ logical operators AND, OR, NOT
→ Relational operators <, <=, >, >=, =, !=

7) The following looping statements are employed.

For while and repeat-until

While loop:

while <condition> do

{

<statement 1>

⋮

<Statement-n>

}

For loop:

For variable := value-1 to value-2 step step do

{

<Statement-1>

⋮

<statement-n>

}

repeat until:

repeat <Statement-1>

⋮

<Statement-n>

until <condition>

(8) A conditional statement has the following forms.

→ if <condition> then <statement>

→ If <condition> then <statement-1>
Else <statement-1>

Case statement :-

Case
{
:<condition-1> : <statement-1>
:
:
:< condition-n> : <Statement-n>
: else : < Statement-n+1>
}

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

(10) There is only one type of procedure:

Algorithm Name (Parameter lists)

Eg:- Algorithm to find max of two numbers

```
algorithm max (A,n)
  // A is an array of size n.
  {
  Result := A[1];
   for I := 2 to n do
     if A[I] > Result then
       Result := A[I];
  return Result;
  }
```

Algorithm for selection sort:-

```
Algorithm selection sort (a,n)
// sort array a[1:n] into non-decreasing
order.
{
  for i:= 1 to n do
     j:= i;
     for k:= i+1 to n do
        if (a[k] < a[j]) then
           j:= k;
           t:= a[i];
           a[i]:= a[j];
           a[j]:= t;
     }
  }
```

Performance Analysis:-

(1) Space complexity:- It is the total amount of memory space used by an algorithm/program including the space of input values for execution. It is used to calculate the space occupied by the variables used in an algorithm/program.

The program source code has many types of variables and their memory requirements are different.

<u>Fixed variables</u>) -

The fixed part of the program are the instructions, simple variables, constants that does not need much memory and they do not change during execution.

<u>Dynamic variables</u>/ Variable part

The variable depends on input size, pointers that refers to other variables dynamically, stack space for recursion.

It is denoted as

$$S_{(P)} = C + S_P(I)$$

$S_P(I) \rightarrow$ instance characteristics

$S_{(P)}$ ↑ problem

C ↑ static fixed variable

S_P ↑ dynamic variable.

<u>Note</u>:- we concentrate only on measuring the space required for dynamic part.

$$\boxed{\text{space complexity } S(P) = C + Sp(I)}$$

where C - fixed space requirements (constants)

$Sp(I)$ = variable space requirements.

Space complexity refers to the worst case (O) and denoted as an asymptotic expression in size of input.

O(n) :- space algorithm requires a constant amount of memory for input.

O(1) :- space algorithm requires a constant amount of space independent of size of input.

Algorithm Sum(a, n)
```
{
    s := 0.0;
    for i := 1 to n do
    s := s + a[i];
    return s;
}
```

For a program
```
#include<stdio.h>
int main()
{
int a=5, b=5, c;
c= a+b;
printf("%d", c);
}
```
variables are a, b, c
int will occupy 4 bytes
so $4 \times 3 = 12$ bytes.

variables are s, i, n, a[ ].

each variable will occupy one space of memory.

s=1, i=1, n=1

a[ ] is an array variable it requires n words of space that holds 'a' must hold for n elements to be sumed.

The total space occupied is n+3

Perfermence Analysis:-

The evaluation can be done in two ways
① priori Estimates | performance Analysis
② Posteriori Testing | performance Meaurement.

Priori	posteriori
(1) The time taken for executing the algorithm is analyzed prior to the execution of algorithm. i.e before running on the system checking the space and time complexity	(1) The execution time taken by an algorithm is evaluated while the algorithm is being executed. i.e After running on the system analysis is made. i.e time & space complexity.
(2) It is also called as performance analysis that evaluates whether the code is readable or it performs the desired functions	(2) It is also called as performence meaurement that measures the accuracy (exact) time & space of algorithm.
(3) It focuses on determining the order of execution of statement.	(3) It focuses on determining the time & space complexity of particular algorithm
(4) It provides approximate values	(4) It provides accurate values.
(5) It is very expensive i.e depends upon the system which has been used for execution	(5) It is very less expensive manually calculated.
(6) Use the Asymptotic notations	(6) Directly depends on system and changes from system to system.

The performance analysis of any algorithm is calculated using two types of complexities

(1) Space complexity
(2) Time complexity.

Time complexity :- The time complexity is the amount of the compute time it needs to run for completion, i/e sum of compile time and run time (execution)

(or)

The time complexity is the number of operations an algorithm performs to complete its task (considering that each operation takes the same amount of time).

→ Algorithm that performs the task in the smallest no. of operations is considered the most efficient one in terms of the time complexity.

The time T(P) taken by a program P is the sum of the compile time and the run time (execution).

→ run time is denoted by t_p (instance characteristics)

$$t_p(n) = C_a\, ADD(n) + C_s\, SUB(n) + C_m\, MUL(n) + C_d\, DIV(n) + \ldots$$

The time complexity can be expressed in 3 different ways or types of time complexity ~~dari~~

1. count method
2. Frequency method
3. Asymptotic notation method.

Count Method:-

→ We introduce a new variable count into the program, it is a global variable with initial value 0.

→ Each time a statement in the original program is executed count is incremented by the step count of that statement.

```
Algorithm sum(a,n)
{
    s := 0.0  // count := count+1;
    for i := 1 to n do  // count := count+1;
    {
        s := s+ a[i]; // count := count+1;
    }  // count := count+1;
    return s; // count := count+1;
}
```

2n

3

2n+3   Total time complexity is 2n+3

17

Algorithm Add (a, b, c, m, n)

{

for i := 1 to m do // count := count + 1;

{

for j := 1 to n do // count := count + 1;

{

c[i,j] := a[i,j] + b[i,j]; // count := count + 1

} // count := count + 1;

} // count := count + 1

}

2m

2mn

1

Total time complexity is $2mn + 2m + 1$

Algorithm MUL (a, b, c, m, p, n)

{

for i := 1 to m do

{ // c := c + 1

for j := 1 to p do // c = c + 1

{

c[i,j] := 0; // c := c + 1

for k := 1 to n do c := c + 1

{

c[i,j] := c[i,j] + a[i,k] * b[k,j]; // c := c + 1

} // c = c + 1

} c = c + 1

} c = c + 1

} // c := c + 1

2m

3mp

2mnp

Total time complexity is $2mnp + 3mp + 2m + 1$

Frequency Method:- Which is to be determine the step count of an algorithm is to build a table in which we list the total no. of steps contributed by each statement.

1st column → statement in which create the algorithm of a given problem.

2nd column → s/e, which indicate steps for execution of the statement.

3rd column → is frequency which indicates the total no. of times (frequency) each statement is executed.

4th column → total steps that is s/e * frequency

Statement	s/e	frequency	Total steps = s/e * frequency
Algorithm Sum(a,n)	0	0	0
{	0	0	0
S := 0.0;	1	1	1
for i := 1 to n do	1	n+1	n+1
S := S+a[i];	1	n	n
return s;	1	1	1
}	0	0	0

$1+n+1+n+1 = 2n+3$ Total Time complexity is $\boxed{2n+3}$

Statement	s/e	frequency	total steps
Algorithm add(a,b,c,m,n)	0	0	0
{	0	0	0
for i := 1 to m do	1	m+1	m+1
{	0	0	0
for j := 1 to n do	1	m(n+1)	mn+m
{	0	0	0
c[i,j] := a[i,j] + b[i,j].	1	mn	mn
}	0	0	0
}	0	0	0
}			

Time complexity is $m+1+mn+m+mn$

$= 2m+2mn+1$

The order of seven computing times are $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(n^3)$, $O(2^n)$.

→ $O(1)$ = constant

→ $O(n)$ = linear

→ $O(n^2)$ = quadratic

→ $O(n^3)$ = cubic

→ $O(2^n)$ = Exponential.

Asymptotic notations

→ The efficiency of an algorithm depends on the amount of time, storage and other resources required to execute the algorithm.

→ The efficiency is measured with the help of asymptotic notations.

→ The study of change in performance of the algorithm with the change in the order of the input size is defined as asymptotic analysis.

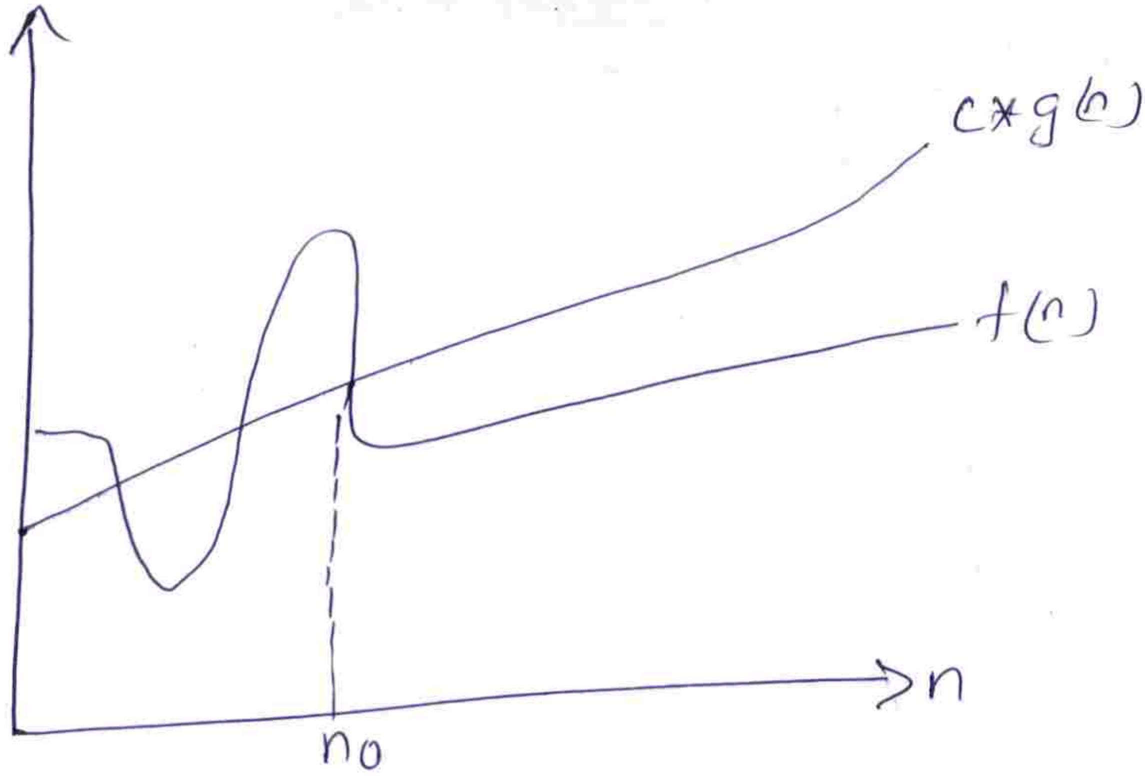
→ Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.

There are 3 types of asymptotic notations

1. Big - O notation
2. Omega notation (Ω)
3. Theta notation (θ)

Big-O notation:- It represents the upper bound of the running time of an algorithm.

→ It gives the worst-case complexity of an algorithm.



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

The function $f(n) = O(g(n))$ "read as f of n is big oh of g of n" iff there exists positive constants c and n_0 such that

$\boxed{f(n) \leq c * g(n)}$ for all n, $n \geq n_0$.

Following the steps to calculate 'O' for a program.

(1) Break the program into smaller segments.

(2) Find the no. of operations performed for each segment (in terms of the input size) assuming the given input is such that the program takes the maximum time i.e the Worst-case scenario

(3) Add up all the operations and simplify it lets say it is f(n).

(4) Remove all the constants and choose the terms having the highest order because for n tends to infinity the constants and the lower order terms in $f(n)$ will be insignificant, let say the function is $g(n)$ then big-O notation is $O(g(n))$

Eg:- ① $3n+3 = O(n)$

$f(n) \leq c * g(n)$

$f(n) = 3n+3$

$g(n) = O(n)$

$3n+3 \leq c * n$

$3n+3 \leq 4n$ now find the 'n' values

$n=1$	$n=2$	$n=3$
$3+3 \leq 4$	$3(2)+3 \leq 4(2)$	$3(3)+3 \leq 4(3)$
$5 \leq 4$ ✗	$6+3 \leq 8$	$9+3 \leq 12$
	$9 \leq 8$ ✗	$12 \leq 12$ ✓

$n \geq 3$ $\quad n_0 = 3$

② $10n^2+4n+2 = O(n^2)$

$f(n) = 10n^2+4n+2$

$g(n) = n^2$

$f(n) \leq c * g(n)$

$10n^2 + 4n + 2 \leq c * n^2$

$10n^2 + 4n + 2 \leq 11n^2$

$n = 1$ $16 \leq 11$ ✗

$n = 2$ $50 \leq 44$ ✗

$n = 3$ $104 \leq 99$ ✗

$n = 4$ $178 \leq 176$ ✗

$n = 5$

$275 \leq 275$ ✓

$n \geq 5$

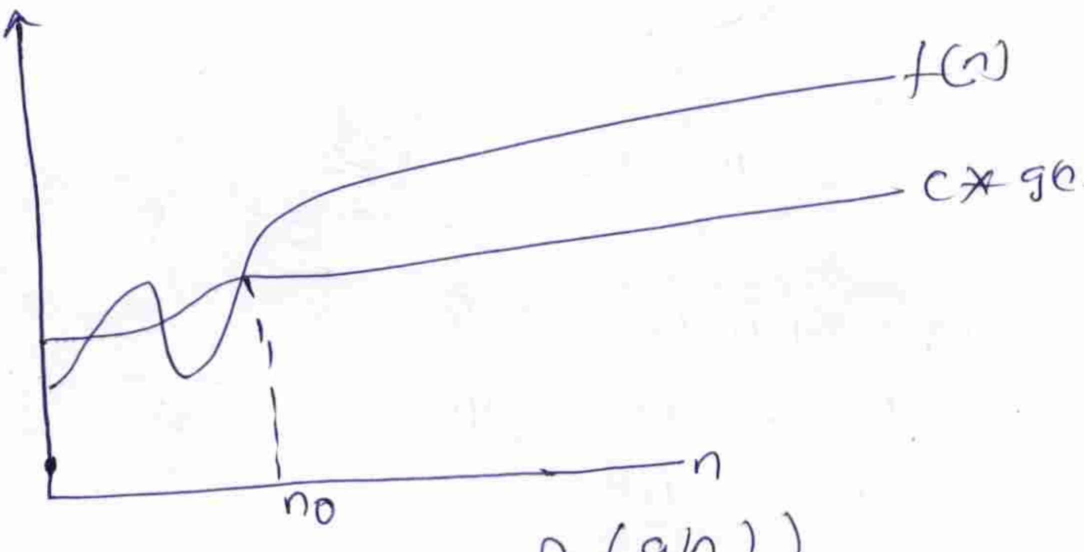
(3) $6 * 2^n + n^2 = 2^n$

(4) $100n + 6 = O(n)$

(5) $3n + 2 = O(n)$

<u>Omega notation (Ω)</u> :- It represents the lower bound of the running time of an algorithm.

→ It provides the best case complexity of an algorithm.



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

The function $f(n) = \Omega(g(n))$ read as f of n is Omega of g of n if and only if there exists positive constants c and no such that

$\boxed{f(n) \geq c * g(n)}$ for all n, $n \geq no$.

To calculate Ω for a program

1. Break the program into smaller segments.
2. Find the number of operations performed for each segment in terms of the input size assuming the given input is such that the program takes the least amount of time.
3. Add up all the operations and simplify it, let's say it is $f(n)$.
4. Remove all the constants and choose the term having the least order or any other function which is always less than $f(n)$ when n tends to infinity, let say it is g(n) Omega (Ω), $f(n)$ is $\Omega(g(n))$

Note: Omega notation does not really help to analyze an algorithm because it does not consider to evaluate an algorithm for the best cases of inputs

Eg:-

① $3n+2 = \Omega(n)$

$f(n) = 3n+2$

$g(n) = n.$

$f(n) \geq c * g(n)$

$3n+2 \geq c * n$

$3n+2 \geq 3n$

$n=1$

$5 \geq 3$ ✓

$n \geq 1$

② $3n+3 = \Omega(n)$

$3n+3 \geq c * n$

$3n+3 \geq 3n$

$n=1$

$6 \geq 3 * 1$

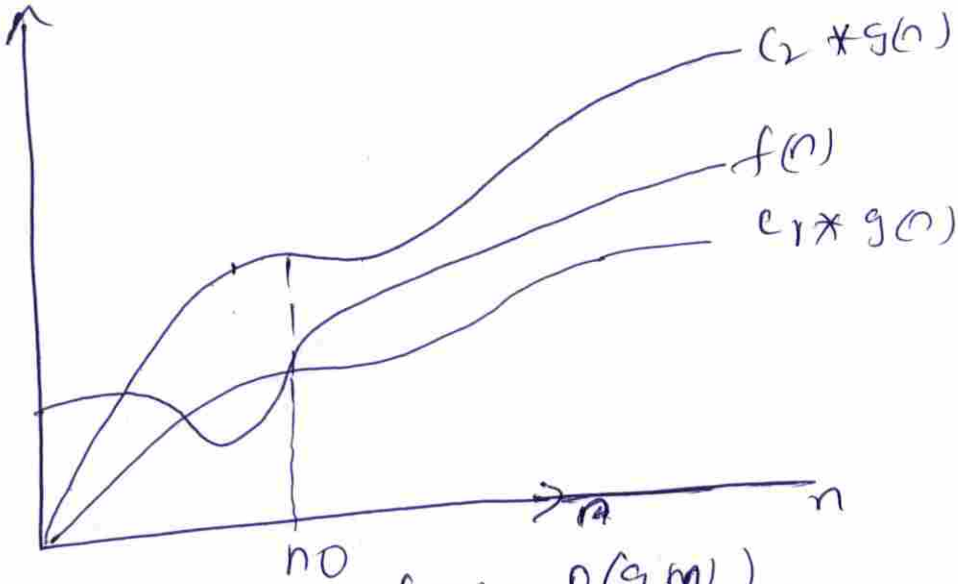
$6 \geq 3$ ✓

$n \geq 1$

According to definition of omega (Ω), the n_0 value should be greater than zero always.

Theta notation:- (Θ) :- Big theta (Θ) notation specifies a bound for a function $f(n)$.

$f(n) = \Theta(g(n))$ read as f of function n is theta of g of n if and only if there exists positive constant c_1, c_2 & n_0 such that $\boxed{c_1 * g(n) \leq f(n) \leq c_2 * g(n)}$ for all n, $n \geq n_0$.



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

Theta notation encloses the function from above and below. since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.

~~$C_1 * g(n)$~~

$C_1 * g(n) \leq f(n) \leq C_2 * g(n)$

(lower ~~upper~~bound value) Ω omega

(upperbound value.) O Big-O

for all n, $n \geq n_0$.

steps to calculate $\underline{\theta}$ for a program

① Break the program into smaller segments.

② Find all types of inputs and calculate the no. of operations they take to be executed. Make sure that the input cases are equally distributed.

③ Find the sum of all the calculated values and divide the sum by the total no. of inputs.

let say the function of $\underline{n}$ obtained is $\underline{g(n)}$ after removing all the constants, then in $\underline{\theta}$ notation its represented as $\underline{\theta(g(n))}$

Eg:- $10n^2 + 4n + 2 = \theta(n^2)$

$f(n) = 10n^2 + 4n + 2$

$g(n) = n^2$

$c_1 * n^2 \leq 10n^2 + 4n + 2 \leq c_2 * n^2$

$10n^2 \leq 10n^2 + 4n + 2 \leq 11n^2$

$n=1$ $10 \leq 16 \leq 11$ ×

$n=2$ $40 \leq 50 \leq 44$ ×

$n=3$ $90 \leq 104 \leq 99$ ×

$n=4$ $160 \leq 178 \leq 176$ ×

$n=5$ $250 \leq 272 \leq 275$ ✓

$\underline{n \geq 5}$

Little "oh" notation:- The function $f(n) = o(g(n))$

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

Little omega notation (ω) :- The function $f(n) = \omega(g(n))$

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

Asymptotic notation used to find the time complexity

Statement	s/e	frequency	Total steps = s/e × frequency
Algorithm sum(a,n)	0	0	$\theta(0)$
{	0	0	$\theta(0)$
s := 0.0;	1	1	$\theta(1)$
for i := 1 to n do	1	n+1	$\theta(n+1)$
s := s + a[i];	1	n	$\theta(n)$
return s;	1	1	$\theta(1)$
}	0	0	$\theta(0)$

As per Asymptotic notation constant values are neglected. ∴ $\theta(0)$, $\theta(1)$ are neglected.

$= \theta(n) + \theta(n+1) + \theta(1) + \theta(1)$

$= \theta(n) + \theta(n) + \theta(1)$ [$\because$ $\theta(1)$ are neglected]

$= \theta(n) + \theta(n)$ [$\because$ $\theta(1)$ are neglected]

$= 2\,\theta(n)$ [$\because$ 2 is constant so neglected]

$= \underline{\underline{\theta(n)}}$

Statement	S/e	Frequency	Total steps
Algorithm Add (a, b, c, m, n)	0	$\theta(0)$	$\theta(0)$
{	0	$\theta(0)$	$\theta(0)$
for i := 1 to m do	1	m+1	$\theta(m+1)$
for j := 1 to n do	1	m(n+1)	$\theta(mn+m)$
c[i, j] := a[i, j] + b[i, j];	1	mn	$\theta(mn)$
}	0	$\theta(0)$	$\theta(0)$

$\theta(m+1) + \theta(mn+m) + \theta(mn)$

$= \theta(m) + \theta(1) + \theta(mn) + \theta(m) + \theta(mn)$

$= \theta(m) + \theta(1) + \theta(mn)$ [$\because$ $\theta(1)$ is neglected]

$= 2\,\theta(m) + 2\,\theta(mn)$

$= \theta(m) + \theta(mn)$ [$\because$ 2 is constant neglected]

$\boxed{= \theta(mn)}$ [$\because$ $mn > m$]

Statement	s/e	Frequency	Total steps
Algorithm mul(a,b,c,m,n)	0	0	$\theta(0)$
{	0	0	$\theta(0)$
for i := 1 to m do	1	$m+1$	$\theta(m+1)$
for j := 1 to p do	1	$m(p+1)$	$\theta(mp+m)$
c[i,j] := 0.0;	1	mp	$\theta(mp)$
for k := 1 to n do	1	$mp(n+1)$	$\theta(mpn+mp)$
c[i,j] := c[i,j] + a[i,k] * b[k,j];	1	mpn	$\theta(mpn)$
}	0	0	

$$= \theta(m+1) + \theta(mp+m) + \theta(mp) + \theta(mpn+mp) + \theta(mpn)$$

$$= \underline{\theta(m)} + \theta(1) + \underline{\theta(mp)} + \underline{\theta(m)} + \underline{\theta(mp)} + \theta(mpn) + \underline{\theta(mp)} + \theta(mpn)$$

$$= 2\theta(m) + 3\theta(mp) + 2\theta(mpn) \quad [\because \theta(1) \text{ is neglected}]$$

$$= \theta(m) + \theta(mp) + \theta(mpn) \quad [\because 2 \text{ is neglected constant}]$$

$$= \underline{\underline{\theta(mpn)}} \quad [\because mpn > mp > m \;\; \therefore \underline{mpn \text{ is considering}}]$$