ANALYSIS OF ALGORITHMS

Algorithm :-

An algorithm is a step by step(or) finite sequence of instructions for solving an instance of a problem or a problem. We can call it as blueprint to write a program.

Once we have an idea or blueprint of a solution, we can implement in any high-level language like C, C++, Java etc..

⇒ Approaches to designing an algorithm :-

(i) Top-Down approach

(ii) Bottom-up approach

(i) Top-Down approach :-

Top down approach starts by dividing the complex algorithm into one or more modules. These modules can further be decomposed into one or more submodules until desired level is achieved.

(ii) Bottom-up approach :-

Bottomup approach is just reverse of a topdown approach. we start with most basic concrete modules and then proceed towards designing higher level modules. sub modules are grouped together to form a higher level module.

⇒ Control structures :-

An algorithm may follow one of the following control structures : Sequence
Decision
Repetition.

Sequence :- Each step of an algorithm is executed in a specific order.
Ex:- Addition of two numbers.

Decision :- Execusion of a process depends on the outcome of some condition. Ex:- IF condition then process 1
else process 2

Repetition :- Executing one or more steps for a number of times, can be implemented using constructs such as while, do-while and for loops.

Analysis of Algorithms → Efficiency of Algorithm:

One problem can have no of algorithms. One should be selected among them.

The performance of algorithm is measured on the basis of following properties:

1. Time complexity
2. Space complexity.

1. Time complexity :- The amount of time taken by an algorithm to run, as a function of the length of the input. It measures the time taken to execute each statement of code in an algorithm.

2. Space complexity :- The amount of (computer) memory (space) an algorithm takes in terms of amount of input to the algorithm.

Space complexity depends on two parts :-

(i) Fixed part (ii) variable part

(i) @ Fixed part varies from problem to problem
(b) Memory required to store instructions, variables and structured variables

(ii) @ It varies from program to program
(b) Space needed for recursion, for structured variables that allocate space dynamically during runtime of a program.

So, while executing programs space is needed to store Instructions – No. of lines in program

Dataspace – Require to store all the constants & variable values

Environment space – To store the environment information needed to resume the suspended function.

There are two types of space complexities.

(i) Constant space complexity:- Here, program required fixed amount of time all input values. Program doesn't contain loops. ex:-

```
int sum(a, b)
{
return a+b;
}
```

⇒ Space needed for only two variables for adding them

Notation = O(1)

(ii) Linear space complexity:- Here, input values increase when loops occur in program.

Notation = O(N)

But, space / memory can be reused easily. Available space can be expanded easily. So, Time complexity is more important than space complexity.

The time complexity of an algorithm can be computed either by empirical (or) theoritical approach.

① Empirical or posteriori testing:-

Executing algorithms on a computer for various instances of the problem and compare the taken for the execution of the programs for various instances. The algorithm which takes the least time, is considered as the best among the algorithms.

② Theoritical or apriori algorithm testing approach:-

In this approach checks the resources that are time and space by mathematical approach. parameters used is the size of input instances.

Ex:- Searching a name in telephone dictionary. If the size is more the time will increase.

Disadvantages of posteriori:-

It is dependent on various factors such as

→ Machine on which program executed
→ Programming language used.
→ Skills of programmer who wrote the code.

Advantages of apriori:-

It is independent to machine, language and program.

Apriori Analysis:-

Apriori analysis is interested in the following for the computation of efficiency:

✓(i) The no. of times the statement is executed in the program, known as frequency count of the statement.

(ii) The time taken for a single execution of the statement.

But, 2nd condition is dependent on machine, so, we are not considering this as main condition.

Let us estimate the frequency count of the statement $x=x+2$ occuring in the following three program segments (A, B, C):

A

```
---
x=x+2 ✓
---
```

B

```
---
for k=1 to n do
  x=x+2 ✓
end
---
```

C

```
---
for j=1 to n do
  for x=1 to n do
    x=x+2 ✓
  end
end
```

- Frequency count of the statement in segment A is 1
- Frequency count of the statement in segment B is n
- Frequency count of the statement in segment C is n^2

Segment A

Program Statements	Frequency count
...	
x = x+2	1
...	
Total frequency count	1

Segment B

Program Segments	Frequency count
...	
for K=1 to N do	n+1
x = x+2	n
end	n
...	
Total frequency count	3n+1

Segment C

program Statements	Frequency Count
...	
for j=1 to n do	n+1
for K=1 to n do	$\sum_{j=1}^{n}(n+1) = n(n+1)$
x = x+2	n^2
end	$\sum_{j=1}^{n}(n) = n^2$
end	n
...	
Total Frequency count	$3n^2+3n+1$

In program,

→ For loop executes ⇒ For i = low-index to up-index

executes = ((upindex - lowindex + 1) + 1) times

= ((n - 1 + 1) + 1)

= n+1 times.

→ Statements in for loop executes

= (upindex - lowindex) + 1

= (n - 1) + 1

= n times

These notation mean that the orders of magnitude of the total frequency counts are proportional to 1, n and n^2 respectively.

Asymptotic Analysis:-

In mathematical analysis, Asymptotic analysis of algorithm is a method of defining the mathematical boundaries of its run-time performance. Asymptotic analysis of algorithm is to estimate the time complexity function for arbitrarily large input. Using the asymptotic analysis, we can easily estimate about the average case, best case and worst case scenario of the algorithm.

Time complexity is a computational way to show how run-time of a program increases as the size of its input increases.

Best case — Best (less) time in which algorithm executes.

Average case — Average time in which algorithm executes.

Worst case — Maximum (worst) time in which algorithm executes.

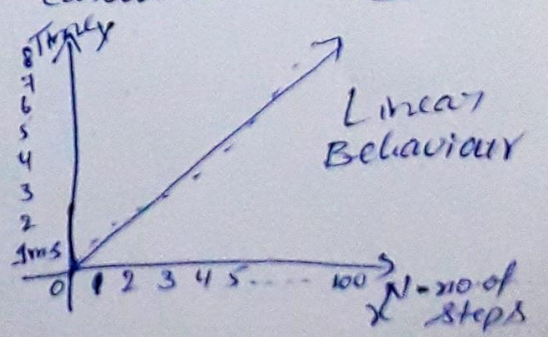
Here, we are checking the time is increasing linearly or exponentially based on increasing the size of instructions

For ex:- Two algorithms to calculate sum of 'N' no's

① function SON(N)

```
sum=0                    → 1ms
for i=0 to n do          → 1ms
    sum=sum+i            → n+1 sec
print sum                → 1m sec
end
```

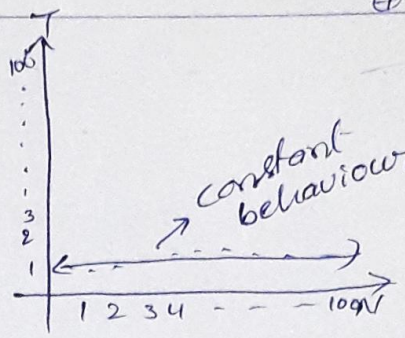
T = n+3msec



② Algorithm 2 (For ex) ↓

```
function SON(N)        → 1M.S
    sum = (N*(N+1))/2  → 2M.S
    print(sum)         → 1M.S
end
```

4M.seconds



In the second algorithm, if we give N=100000000 also, it will give output in fraction of second. But, Algorithm 1 takes no of seconds to give output.

So, Algorithm 2 is much better than Algorithm 1 Because, input increases the time will increase in Algorithm 1.

Asymptotic Notations:-

These notations are defined in terms of functions whose domains are the set of natural numbers $N=\{0,1,2--\}$. Notations can be listed as : O, Ω, Θ, o, ω

1. Big oh notation (O):-

The Big O notation defines an upper bound of an algorithms running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.

The Big O notation, where O stands for 'order of', is concerned with what happens for very large values of N. When expressing complexity using Big O notation, constant multipliers are ignored. So, O(4n) is equivalent to O(n).

If f(n) and g(n) are the functions defined on a positive integer number n, then $\boxed{f(n)=O(g(n)}$

that is, f of n is Big-o of g of n, if and only if

positive constants c and n exist, such that

$$f(n) \leq cg(n).$$

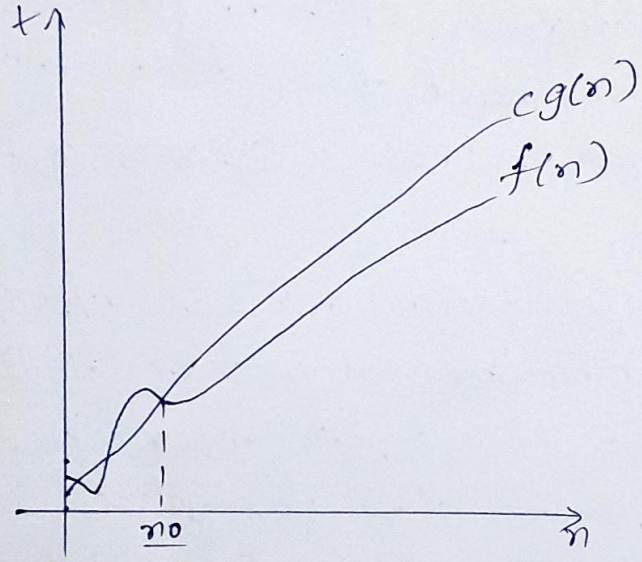
It means that for large amounts of data, $f(n)$ will grow no more than a constant factor than $g(n)$.

Note:- 'c' is a constant which depends on the some factors, ⓐ Programming language used ⓑ quality of compiler ⓒ CPU speed ⓓ size of main memory and access time ⓔ knowledge of programmer ⓕ and algorithm itself.

If $f(n) \leq cg(n)$, $c>0$ for all

$n \geq n_0$ and $n_0 >= 1$.

then $f(n) = O(g(n))$ and $g(n)$ is an asymptotically tight upper bound for $f(n)$.



n = no. of input instructions
t = time

After, some no. of input steps (n_0), $cg(n)$ is always greater than $f(n)$.

Let $f(n) = 2n+3$, $g(n) = n$ $\Rightarrow$ $\boxed{f(n) \leq c \cdot g(n)}$

$2(n)+3 \leq 5.(1)$ Let $c=5, n=1$

$5 \leq 5$ ✓

$2(2)+3 \leq 5(2)$ $c=5, n=2$

$7 \leq 10$ ✓

So, After $n=1$ ($n_0=1$), the $cg(n)$ is always a tight upper bound than $f(n)$.

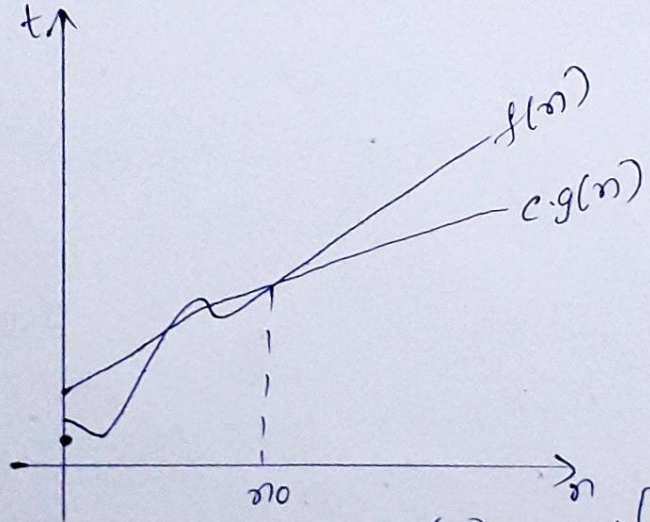
(2) Omega Notation (Ω):-

The Ω notation defines lower bound of an algorithms running time. It measures the best case time complexity or the ~~(shortest)~~ least amount of time an algorithm can possibly take to complete.

The Omega notation provides a tight lower bound for $f(n)$. Ω notation is simply written as, $\boxed{f(n) = \Omega(g(n))}$ Here f and g are functions of n. That is, f of n is Omega of g of n, if and only if positive constants c and n_0 exist, such that

- $f(n) \geq c \cdot g(n)$, $c > 0$ for all $n \geq n_0$ and $n_0 \geq 1$.

Then, $f(n) = \Omega(g(n))$ and $g(n)$ is an asymptotically tight lower bound for $f(n)$.



Let $f(n) = 2n+3$, $g(n) = n$ $\Rightarrow \boxed{f(n) \geq c \cdot g(n)}$

$2(1)+3 \geq 1 \cdot 1$ — let $c = 1$,

$5 \geq 1$ ✓ — $n = 1$

$2(2)+3 \geq 1 \cdot 2$ — $n = 2$

$7 \geq 2$ ✓

For all values of n greater than or equal to n_0, i.e, $n_0 = 1$, $c(g(n))$ is less than $f(n)$.

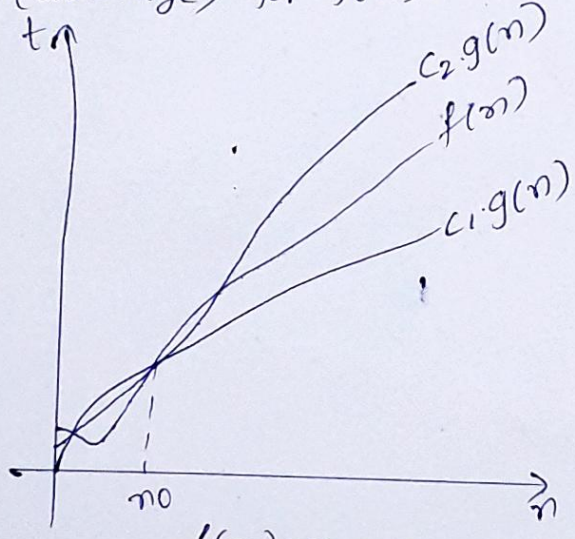
3. Theta Notation (Θ):-

The theta notation defines an average case of an algorithm's running time. It measures the average case i.e; most realistic time complexity of an algorithm.

If $f(n)$ and $g(n)$ are functions defined on a positive integer number n, then $\boxed{f(n) = \Theta(g(n))}$, If and only if $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$, positive constants c_1, c_2 and n exist.

If $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$, $c_1 > 0$, $c_2 > 0$ for all $n \geq n_0$ and $n_0 \geq 1$.

then $f(n) = \Theta(g(n))$ and $g(n)$ is asymptotically tight bound (average) for $f(n)$.



$f(n) = 2n+3$ $\quad g(n) = n$

$1 \cdot 1 \leq 2(1)+3 \leq 5 \cdot 1$

$1 \leq 5 \leq 5$ ✓

$1 \cdot 2 \leq 2(2)+3 \leq 5 \cdot 2$

$2 \leq 7 \leq 10$ ✓

• $c_1 = 1$, $c_2 = 5$ } $n = 1$

$n = 2$

Forall $n_0 \geq 1$, if $c_1 = 1$, $c_2 = 5$ the $\Theta(g(n))$ is always average case scenario of $f(n)$.

There are other useful notations like little o (o) and little omega (ω) notation.

Little o Notation:- It provides a non-asymptotically tight upperbound for $f(n)$. (i.e., loose upper bound). So, $\boxed{f(n) = o(g(n))}$ if and only if $f(n) < g(n)$ where f and g are functions of n, c and n are constants.

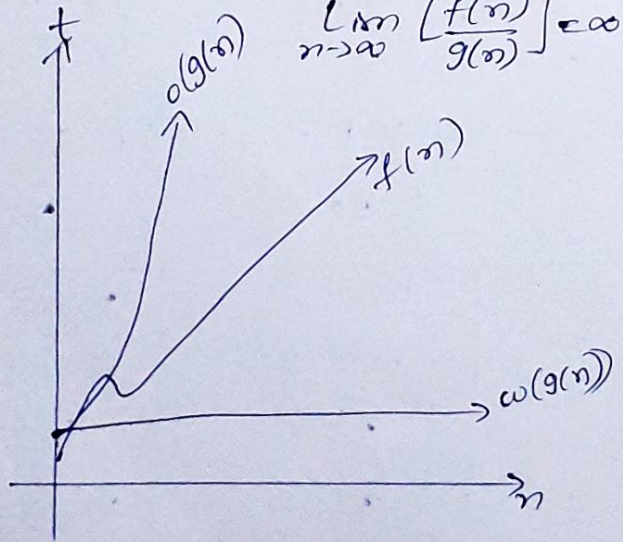
$$f(n) = o(g(n)) \Rightarrow f(n) < c \cdot g(n) \quad c > 0, n \geq n_0, n_0 \geq 1$$

Here, $f(n)$ is smaller order than $g(n)$; i.e., $f(n)$ is small compared to $g(n)$. $\lim_{n \to \infty} \left[\frac{f(n)}{g(n)}\right] = 0$.

ω-notation (small omega):- It provides a non-asymptotically tight lower bound for $f(n)$. (i.e., loose lowerbound). So, $\boxed{f(n) = \omega \cdot g(n)}$ if and only if $f(n) > c \cdot g(n)$ where f and g are functions of n, c and n are constants.

$$f(n) = \omega g(n) \Rightarrow f(n) > c \cdot g(n) \text{ where } c > 0, n \geq n_0 \text{ and } n_0 \geq 1.$$

Here, $g(n)$ is lowerbound for $f(n)$, i.e., loose lower bound. $g(n)$ is smaller than $f(n)$.



Polynomial vs Exponential algorithms :-

⇒ Polynomial $O(n^K)$
where n is the input size,
K is constant

⇒ Exponential K^n or n^n , If $K=2$

For $n=$	2	10	20	30
$2^n =$	4	1024	1 Million	1000 million

Polynomial :-
The algorithm is said to be solved in polynomial time if the no of steps required to complete the algorithm for a given input is $O(n^K)$ for some non-negative (positive) integer K where n is complexity of the input. Basic addition, subtraction, multiplication, division, comparison, sort operations are considered as polynomial time algorithms.

Exponential :-
The problem which can be solved by an exponential time algorithms, is known as exponential ~~time~~ running time. Here, no of steps required is $O(K^n)$, n & K are positive integers.

So, polynomial algorithms are lot more efficient than exponential algorithms.

Comparison of Polynomial and exponential algorithms:

Size / Time complexity function	10	20	50
n^2	10^{-4} sec	4×10^{-4} sec	25×10^{-4} sec
n^3	10^{-3} sec	8×10^{-3} sec	125×10^{-3} sec
2^n	10^{-3} sec	1 sec	35 years
3^n	6×10^{-2} sec	58 min	2×10^3 centuries

Average, Best and worst case complexities:-

The time complexity of an algorithm is dependent on parameters input/output instances of the problem.

For ex:- The process is to find first occurance of even integer on the given list.

Input 1: -1 3 5 7 -5 11 -13 17 71 21 9 3 1 5 6

Input 2: 6 -5 11 3 5 7 12 17 21 9 71 3 5 1 6

Input 3: 71 21 9 3 1 5 -23 36 17 12 7 5 3 11 -5

For input 1, it took the n comparisons because the even number is at nth position. For input 2, it tooks one comparison and for input 3 it tooks average time of the given list. Based on the input, the time is changing.

Average case:- All the input instances which are neither best case nor worst case are categorized average cases and the time complexity of the algorithm in such cases is referred to as the average case time complexity.

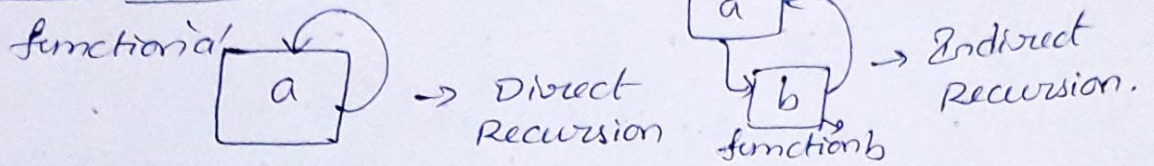
Best case:- The input instances for which the algorithm takes the minimum possible time is called the best case and the time complexity in such a case is referred to as the best case time complexity.

Worst case:- The input instances for which the algorithm takes the maximum possible time is called the worst case and the time complexity in such a case is referred to as the worst case time complexity.

⇒ Analysing Recursive Programs :-

Recursive programs / procedures :-

If a procedure (function) 'a' containing a call statement to itself is called Direct recursion, another call function 'a' containing a call statement to another function 'b' that result in a call to it self 'a' is called as Indirect recursion.



Recursive functions may run into infinite loop. So, it is essential that the following properties are satisfied by any recursive procedures.

(i) There must be criteria, one or more, called 'base criteria' or simply base case(s), where the procedure does not call itself directly or indirectly.

(ii) Each time the procedure calls itself directly or indirectly, it must be closer to the base criteria.

Ex:- A recursive procedure to complete factorial of number 'n' is

$n! = 1$ → if $n = 1$ (base condition)

$n! = n.(n-1)!$ → if $n > 1$

Here, factorial of 'n!' calls $(n-1)!$ for its definition.

```
function factorial(n)
    if (n=1) then factorial=1.
    else
    factorial = n * factorial(n-1).
    end factorial
```