ALGORITHMS.

→ Definition of algorithm is a "formally defined procedure for performing some calculation.

→ If a procedure is formally defined, then it can be implemented using a formal language, and such a language is known as PROGRAMMING LANGUAGE.

→ Algorithm → provides blueprint to write a program to solve a particular problem

→ Algorithm → mainly used to acheive Software Reuse. That can be implemented in any high level lang. like C, C++ or Java.

→ Algorithm basically a set of instruction that solve a problem. There are not uncommon to have multiple algorithms to tackle the same problem, but the choice of a particular algorithm must depend on the Time and Space complexity of the algorithm.

⇒ ALGORITHM SPECIFICATION ⇐

→ An algorithm is a finite set of instruction that, if followed, accomplishes a particular task. In addition all algorithm must satisfy the following criteria : →

(1). Input → There are zero or more quantities that are externally supplied.

(2) Output → At least one quantity is produced.

(3). Definiteness → Each instruction is clear and unambiguous.

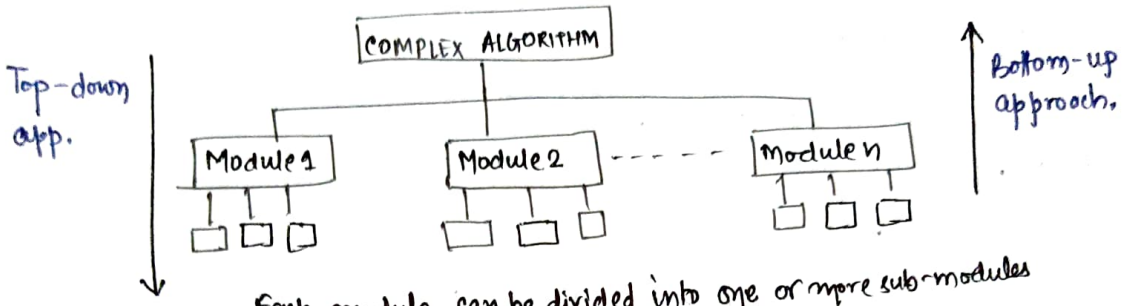
(4). Finiteness → If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.

(5) Effectiveness : → every instruction must be basic enough to be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in (3); it also must be feasible.

In computational Theory, one distinguishes btw an algorithm and a program, the latter of which does not have to satisfy the fourth condition. for example → we can think → of an operating system that continues in a wait loop until more jobs are entered. Such a program does not terminate unless the system crashes. Since our programs will always terminate, we will use algorithm and program interchangeably in this text.

Different Approaches to design an algorithm.

Algorithms are used to manipulate the data contained in data structures. When working with data structures, algorithms are used to perform operations on the stored data.



Each module can be divided into one or more sub-modules

Top-Down Approach.

→ A top-down design approach starts by dividing the complex algorithm into one or more modules. → These modules can further be decomposed into one or more sub-modules & this process of decomposition is iterated until the desired level of module complexity is achieved.

→ Top-down approach → start from a abstract design and then at each step, this design is refined into more concrete levels is reached that requires no further refinement.

→ Top-Down approach → follow a stepwise refinement by decomposing the algorithm into manageable modules.

→ Top-Down approach → high appreciated for ease in documenting the modules, generation of test cases, implementation of code and debugging.

→ Top-down → criticised → because of submodules are analysed in isolation w/o concentrating on their communication with other modules or on reusability of components & little attention paid to data. —

IGNORING THE CONCEPT OF INFORMATION HIDING.

Bottom Up Approach.

- A Bottomup approach is just the reverse of top-down approach.
- designing the most basic or concrete modules and then proceed towards designing higher level modules.
- The higher level modules are implemented by using the operations performed by lower level modules.
- This sub-modules → are grouped together to form a high level modules.
- All the higher level modules are clubbed together to form even higher level modules. This process is repeated until the design of the complete algorithm is obtained.
- Bottomup → app. on other hand defines a modules and then groups together several modules to form a new higher level modules.
- Bottom-up → allows information hiding as it first identifies what has to be encapsulated within a module and then provides an abstract interface to define the modules boundaries as seen from the client.

TIME AND SPACE COMPLEXITY

→ Analysing an algorithm, means determining the amount of resources (such as time and memory) needed to execute it.

→ Algorithm generally designed to work with an arbitrary no. of inputs, so the efficiency or complexity of an algorithm is stated in terms of "TIME & SPACE Complexity".

→ **Time Complexity** → basically the running time of a program as a function of the INPUT SIZE. The no. of machine instruction which a program executes is called Time Complexity. This no. is primarily dependent on the size of the program's input and algorithm used.

→ **Space Complexity** → basically the amount of computer memory that is required during the program execution as a function of the INPUT SIZE.

Generally, the space needed by a program depends on the following two parts:→

(1). Fixed Part:→

→ It varies from problem to problem. It includes the space needed for storing instruction, constants, variables, and structured variables (like array and structure).

(2). Variable Part:→

it varies from program to program. It includes the space needed for recursion stack, & for structured variables that are allocated space dynamically during the runtime of a program.

Worst Case, Average Case, Best Case & Amortized Time Complexity

① Worst Case Time :→ Worst Case running time of an algorithm is an upper bound on the running time for any input. Therefore, having the knowledge of worst-case running time gives us an assurance that the algorithm will never go beyond this time limit.

② Average Case Time :→ It specifies the expected behaviour of the algorithm when the input is randomly drawn from a given distribution. Average case running time assumes that all i/p of a given size are equally like.

③ Best Case Time :→ analyse an algorithm under optimal condition.

④ Amortized Running Time :-

Amortized running time refers to the time required to perform a sequence of related operations averaged over all the operation performed.

Amortized analysis guarantees the average performance of each operation in the worst case.

Time - Space Trade off.

→ Best algo is one which takes less time and less memory to complete its execution.

→ But designing such algorithm is a diff. job.

→ For solving one problem, there can be many ~~problem~~ algorithms, in that some take less time while others take less memory.

→ One may require less memory space other may require less cpu time to execute.

There is always a trade off b/w space and time.

→ So, if space is a big constraint, then one might choose that algo. which take less space.

→ so, if time is a big u, take less time / cpu cycles.

→ The most widely used notation to express this function f(n) is the Big O notations. It provides the upper bound for the complexity.

Expressing Time and Space Complexity

Time and space complexity can be expressed using a function f(n) where n is the input size for a given instance of the problem being solved.

Expressing the complexity is required when :-

→ We want to predict the rate of growth of complexity as the input size of the problem increases.

→ There are multiple algorithms that find a solution to a given problem and we need to find the algorithm that is most efficient.

Space and Time Complexity

→ Time complexity of different Data Structure.

BEST CASE TIME COMPLEXITY OF DIFFERENT DATA STRUCTURES				
DATA STRUCTURE	ACCESS	SEARCH	INSERTION	DELETION
ARRAY	O(1)	O(1)	O(1)	O(1)
STACK	O(1)	O(1)	O(1)	O(1)
QUEUE	O(1)	O(1)	O(1)	O(1)
SINGLY LINKED LIST	O(1)	O(1)	O(1)	O(1)
DOUBLY LINKED LIST	O(1)	O(1)	O(1)	O(1)
HASH TABLE	O(1)	O(1)	O(1)	O(1)
BINARY SEARCH TREE	O(log n)	O(log n)	O(log n)	O(log n)
AVL TREE	O(log n)	O(log n)	O(log n)	O(log n)
B-TREE	O(log n)	O(log n)	O(log n)	O(log n)
RED BLACK TREE	O(log n)	O(log n)	O(log n)	O(log n)

WORST CASE TIME COMPLEXITY OF DIFFERENT DATA STRUCTURE				
DATA STRUCTURE	ACCESS	SEARCH	INSERTION	DELETION
ARRAY	O(1)	O(N)	O(N)	O(N)
STACK	O(N)	O(N)	O(1)	O(1)
QUEUE	O(N)	O(N)	O(1)	O(1)
SINGLY LINKED LIST	O(N)	O(N)	O(N)	O(N)
DOUBLY LINKED LIST	O(N)	O(N)	~~O(N)~~ O(1)	O(1)
HASHTABLE	O(N)	O(N)	O(N)	O(N)
BINARY SEARCH TREE	O(N)	O(N)	O(N)	O(N)
AVL TREE	O(log n)	O(log n)	O(log n)	O(log n)
BINARY TREE	O(N)	O(N)	O(N)	O(N)
RED BLACK TREE	O(log N)	O(log N)	O(log N)	O(log N)

THE AVERAGE TIME COMPLEXITY OF DIFFERENT DATA STRUCTURES				
DATA STRUCTURE	ACCESS	SEARCH	INSERTION	DELETION
ARRAY	O(1)	O(N)	O(N)	O(N)
STACK	O(N)	O(N)	O(1)	O(1)
QUEUE	O(N)	O(N)	O(1)	O(1)
SINGLY LINKED LIST	O(N)	O(N)	O(1)	O(1)
DOUBLY LINKED LIST	O(N)	O(N)	O(1)	O(1)
HASH TABLE	O(1)	O(1)	O(1)	O(1)
BINARY SEARCH TREE	O(log N)	O(log N)	O(log N)	O(log N)
AVL TREE	O(log N)	O(log N)	O(log N)	O(log N)
B TREE	O(log N)	O(log N)	O(log N)	O(log N)
RED BLACK TREE	O(log N)	O(log N)	O(log N)	O(log N)

TIME COMPLEXITY OF DIFFERENT SORTING ALGORITHM

SORTING ALGO.	BEST	AVERAGE	WORST
INSERTION	$\Omega(n)$	$\theta(n^2)$	$O(n^2)$
SELECTION	$\Omega(n^2)$	$\theta(n^2)$	$O(n^2)$
BUBBLE	$\Omega(n)$	$\theta(n^2)$	$O(n^2)$
HEAP	$\Omega(n\log n)$	$\theta(n\log n)$	$O(n\log n)$
QUICK	$\Omega(n\log n)$	$\theta(n\log n)$	$O(n^2)$
MERGE	$\Omega(n\log n)$	$\theta(n\log n)$	$O(n\log n)$
BUCKET	$\Omega(n+k)$	$\theta(n+k)$	$O(n^2)$
RADIX	$\Omega(nk)$	$\theta(nk)$	$O(nk)$
COUNT	$\Omega(n+k)$	$\theta(n+k)$	$O(n+k)$
SHELL	$\Omega(n\log n)$	$\theta(n\log n)$	$O(n^2)$
TREE	$\Omega(n\log n)$	$\theta(n\log n)$	$O(n^2)$

ASYMPTOTIC NOTATION

① BIG O Notation

- If we have two different algorithms to solve the same problem where one algorithm executes in 10 iteration and other in 20 iteration, the difference btw the two algorithm is not much. However if the first algorithm executes in 10 iteration and the other in 1000 iterations, then it is a matter of concern.

- Big O notation, where O stands for 'order of' is concerned with what happen for very large value of n.
  - ↳ Ex → If a sorting algo. performs n^2 operations to sort just n elements, then that algorithm would be described as $O(n^2)$ algorithm.

→ While expressing complexity using the Big O notation, constant multipliers ignored.
  - ↳ Ex → $O(4n)$ algorithm is equivalent to $O(n)$.

→ If $f(n)$ and $g(n)$ are functions defined on a positive integer number n, then $f(n) = O(g(n))$
That is, f of n is Big-O of g of n if and only if positive constant c and n exists, such that $f(n) \leq cg(n)$.

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

→ Hence g provides an upper bound.

→ Big O → provides a strict upper bound for $f(n)$. This means that the function $f(n)$ can do better but not worse than the specified value.

- Big O notation is simply written as $f(n) \in O(g(n))$ or as $f(n) = O(g(n))$

→ Here, n is the problem size and $O(g(n)) = \{ h(n) : \exists$ positive constants c, n_0 such that $0 \leq h(n) \leq cg(n), \forall n \geq n_0 \}$. Hence, we can say that $O(g(n))$ comprises a set of all the functions $h(n)$ that are less than or equal to $cg(n)$ for all value of $n \geq n_0$.

→ Best Case O → describes an upper bound for all combination of input. It is possibly lower than the worst case.

→ Worst Case O → describes a lower bound for worst case i/p combination. It is possibly greater than the Best Case.

Ex → of $f(n)$ and $g(n)$

$g(n)$	$f(n) = O(g(n))$
10	$O(1)$
$2n^3+1$	$O(n^3)$
$3n^2+5$	$O(n^2)$
$2n^3+3n^2+5n-10$	$O(n^3)$

n	O(1)	O(log n)	O(n)	O(nlogn)	$O(n^2)$	$O(n^3)$
1	1	1	1	1	1	1
2	1	1	2	2	4	8
4	1	2	4	8	16	64
8	1	3	8	24	64	512
16	1	4	16	64	256	4096

No. of Operations for different functions of n

Ques 1 Show that $4n^2 = O(n^3)$.

By definition, we have

$$0 \le h(n) \le c\,g(n)$$

Substituting $4n^2$ as $h(n)$ and n^3 as $g(n)$

$$0 \le 4n^2 \le c\,n^3$$

dividing by n^3

$$0/n^3 \le \frac{4n^2}{n^3} \le cn^3/n^3$$

$$0 \le \frac{4}{n} \le c$$

Now to determine the value of c, we see that $4/n$ is maximum when $n=1$. Therefore, $c=4$.

To determine the value of n_0

$$0 \le 4/n_0 \le 4$$

$$0 \le 4/4 \le n_0 \Rightarrow 0 \le 1 \le n_0$$

This mean $n_0 = 1$, Therefore

$$0 \le 4n^2 \le 4n^3, \quad \forall\ n \ge n_0 = 1.$$

Ques 2 Show that $400n^3 + 20n^2 = O(n^3)$

By definition, we have

$$0 \le h(n) \le c(g(n))$$

Substituting $400n^3 + 20n^2$ ~~with~~ as $h(n)$ and n^3 as $g(n)$, we get

$$0 \le 400n^3 + 20n^2 \le cn^3$$

dividing by n^3

$$\frac{0}{n^3} \le \frac{400n^3}{n^3} + \frac{20n^2}{n^3} \le \frac{cn^3}{n^3}$$

$$0 \le 400 + \frac{20}{n} \le c$$

Note that $\frac{20}{n} \to 0$ as $n \to \infty$, and $20/n$ is maximum when $n=1$. Therefore,

$$0 \le 400 + \frac{20}{1} \le [$$

This means $c = 420$

To determine the value of n_0,

$$0 \le 400 + \frac{20}{n_0} \le 420$$

$$0 - 400 \le 400 + \frac{20}{n_0} - 400 \le 420 - 400$$

$$-400 \le \frac{20}{n_0} \le 20$$

$-20 \le 1 \le n_0$. This implies $n_0 = 1$.

Hence, $0 \le 400n^3 + 20n^2 \le 420n^3$

$\forall\ n \ge n_0 = 1.$

Limitation of Big O Notation

→ Big O analysis only tells us how the algorithm grows with the size of the problem, not how efficient it is, as it does not considers the programming effort.

→ It ignores important constants. for example, if one algorithm takes $O(n^2)$ time to execute and the other takes $O(100000n^2)$ time to execute, then as per Big O, both algorithm have equal time complexity. In real-time systems, this may be a serious consideration.

Ex:- Find the upper bound for $f(n) = 3n+8$

$\Rightarrow 3n+8 \leq 4n$

for all value of $n \geq 8$

$\therefore 3n+8 = O(n)$ with $c=4$ and $n_0=8$.

Ex 2:- Find the upper bound for $f(n) = n^4+100n^2+50$

$\Rightarrow n^4 + 100n^2 + 50 \leq 2n^4$

$\frac{0}{n^4} \leq 1 + \frac{100n^2}{n^4} + \frac{50}{n^4} \leq c$

$0 \leq 1 + \frac{100}{n^2} + \frac{50}{n^4} \leq c$

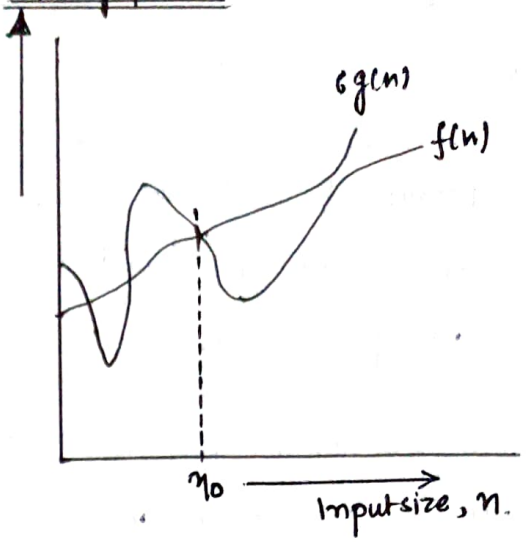
or

$n^4 + 100n^2 + 50 \leq 2n^4$

this is for all $n \geq 11$

$\therefore n^4 + 100n^2 + 50 = O(n^4)$ with $c=2$ & $n_0=11$.

Rate of Growth



At larger value of n, the upper bound of $f(n)$ is $g(n)$. for ex, if $f(n)$, $f(n) = n^4 + 100n^2 + 10n + 50$ is the given algorithm, then n^4 is $g(n)$.

That means $g(n)$ gives the maximum rate of growth for $f(n)$ at larger values of n.

No Uniqueness?

There is no unique set of values for n_0 and c in proving the asymptotic bounds. Let us consider, $100n+5 = O(n)$. For this function there are multiple n_0 and c values possible.

Ex1→ $100n + 5 \leq 100n + n = 101n \leq 101n$ for all $n \geq 5$, $n_0=5$ and $c=101$ is a solution.

Ex→1 $100n + 5 \leq 100n + 5n = 105n \leq 105n$, for all $n \geq 1$, $n_0=1$ and $c=105$ is also a solution.

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

$\Theta \rightarrow$ is most useful than O and Ω, than because why we need upper bound ~~or~~ because if we do not specify any specific position of input thay we need an Upper Bound.

* if we goes for any algorithm the time complexity will be among above mentioned complexity or will be multiple of these complexity and if the time complexity is not among these than we can't exactly represent Θ in that scenerio we required or go for O or Ω.

Ex $f(n) = 2n+3$

$2n+3 \leq 2n+3n$

$2n+3 \leq 5n$, this is true for all $n \geq 1$.

↳ this is $g(n) = n$

$f(n) = O(n)$

same func. $f(n) = 2n+3$

$2n+3 \leq 2n^2+3n^2$

$2n+3 \leq 5n^2$, $n \geq 1$

$g(n) = n^2$

$f(n) = O(n^2)$ this is also true.

So both $O(n)$ and $O(n^2)$ will be upper bound for same function.

if we closely observe that all function after (n) will be upper bound.

average bound

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

lower bound

upper bound,

→ Same function $f(n) = 2n+3$ cant be said $O(\log n)$ because they are coming in the lower bound.

→ when you write $O \rightarrow$ try to write closest function.

(2) OMEGA NOTATION (Ω)

→ provides the tight lower bound for $f(n)$. This means that the function can never do better than the specified value but it may do worse.

→ If $cg(n) \leq f(n)$, $c>0$, $\forall\ n \geq n_0$, then $f(n)$, $f(n) \in \Omega(g(n))$ and $g(n)$ is an asymptotically tight lower bound for $f(n)$.

→ Ex of functions in $\Omega(n^2)$ include: n^2, $n^{2.9}$, n^3+n^2, n^3.

→ Ex of functions not in $\Omega(n^3)$ include: n, $n^{2.9}$, n^2.

→ $\Omega \Rightarrow$ it means best case.

Ex Show that $5n^2+10n = \Omega(n^2)$

By definition, we can write

$$0 \leq cg(n) \leq h(n)$$

$$0 \leq cn^2 \leq 5n^2+10n$$

dividing by n^2

$$0/n^2 \leq cn^2/n^2 \leq \frac{5n^2}{n^2} + \frac{10n}{n^2}$$

$$0 \leq c \leq 5 + \frac{10}{n}$$

Now, $\lim_{n\to\infty} 5+\frac{10}{n} = 5$

Therefore $0 \leq c \leq 5$

Hence, $c=5$

Now to determine the value of n_0

$$0 \leq 5 \leq 5+10/n_0$$

$$-5 \leq 5-5 \leq 5+\frac{10}{n_0} - 5$$

$$-5 \leq 0 \leq 10/n_0$$

So $n_0 = 1$ as $\lim_{n\to\infty} 1/n = 0$

Hence, $5n^2+10n = \Omega(n^2)$ for $c=5$ and $\forall\ n \geq n_0 = 1$.

Ex → Show that $7n \neq \Omega(n^2)$

By definition, we can write

$$0 \leq cg(n) \leq h(n)$$

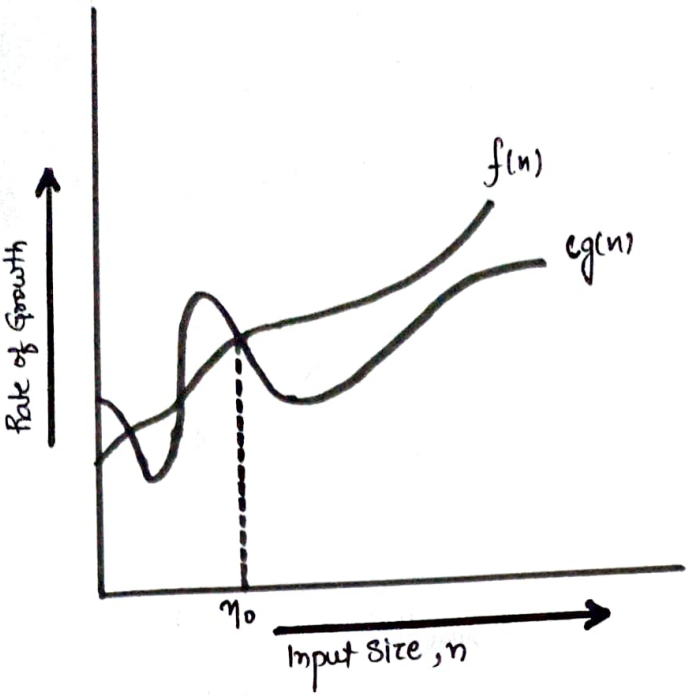
$$0 \leq cn^2 \leq 7n$$

dividing by n^2, we get

$$0/n^2 \leq cn^2/n^2 \leq 7n/n^2$$

$$0 \leq c \leq 7/n$$

Thus, from the above statement, we see that the value of c depends on the value of n. There does not exists a value of n_0 that satisfies the condition as n increase. This could fairly be possible if $c=0$ but it is not allowed as the definition by itself says that $\lim_{n\to\infty} 1/n = 0$.



Ex→ Find the lower bound for $f(n) = 5n^2$

Soln→ $\exists\ c, n_0$ such that : $0 \leq cn^2 \leq 5n^2$

$\Rightarrow cn^2 \leq 5n^2$

dividing by n^2

$c \leq 5$, $c = 5$ and $n_0 = 1$

$\therefore 5n^2 = \Omega(n^2)$ with $c = 5$ and $n_0 = 1$.

Ex→ Prove $f(n) = 100n + 5 \neq \Omega(n^2)$

Soln $\exists\ c, n_0$ such that : $0 \leq cn^2 \leq 100n + 5$

$100n + 5 \leq 100n + 5n\ (\forall\ n \geq 1) = 105n$

$cn^2 \leq 105n \Rightarrow n(cn - 105) \leq 0$

Since n is positive $\Rightarrow cn - 105 \leq 0$

$\Rightarrow n \leq 105/c$

⇒ Contradiction : n cannot be smaller than a constant.

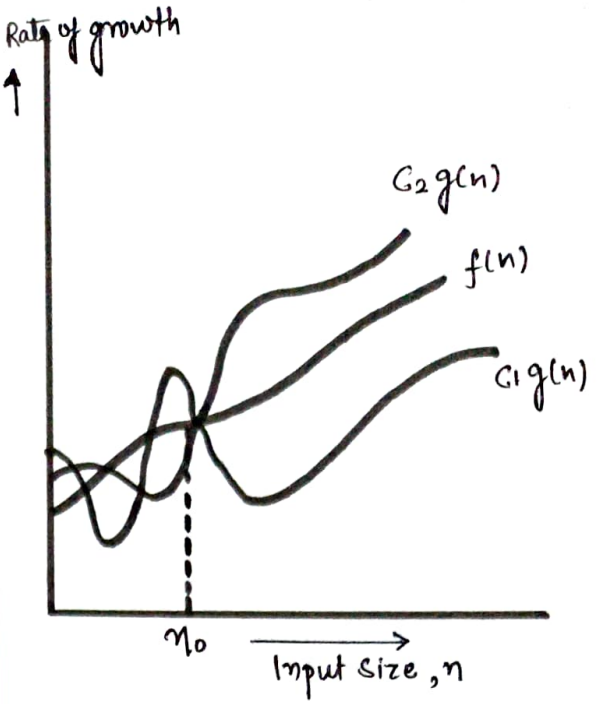
(3) THETA NOTATION (θ)

→ Theta notation provides an asymptotically tight bound for $f(n)$.

→ θ notation is simply written as, $f(n) \in \theta(g(n))$ where n is the problem size and

$\theta(g(n)) = \{ h(n) : \exists$ positive constants c_1, c_2 and n_0 such that $0 \leq c_1 g(n) \leq h(n) \leq c_2 g(n)$, $\forall\ n \geq n_0 \}$.

Hence, we can say that $\theta(g(n))$ comprises a set of all the functions $h(n)$ that are b/w $c_1 g(n)$ and $c_2 g(n)$ for all values of $n \geq n_0$.



→ To summarize,
- The best case is θ notation is not used.
- Worst case θ describe asymptotic bounds for worst case combination of input values.
- If we simply write θ, it means same as worst case θ.

- This notation decides whether the upper and lower bounds of a given function (algo) are the same
- Avg running time of an algorithm is always b/w the lower bound and upper bound.
- If the upper bound (O) and lower bound (Ω) gives the same result, then the θ notation will also have the same rate of growth.

④ Little o Notation

This notation provide a non asymptotically tight upper bound for f(n).

$f(n) \in o(g(n))$ where

$o(g(n)) = \{ h(n) : \exists$ positive constants c, n_0 such that any $c > 0$, $n_0 > 0$ and $0 \leq h(n) \leq cg(n)$, $\forall\ n \geq n_0 \}$

This is unlikely the Big O notation where we say for some $c > 0$ (not any).

for ex $5n^3 = O(n^3)$ is asymptotically tight upper bound but $5n^2 = O(n^3)$ is non-asymptotically tight bound for f(n).

⑤ Little Omega Notation (ω)

→ This notation provides a non-asymptically tight lower bound for $f(n)$.

→ It can be simply written as, $f(n) \in \omega(g(n))$, where

$\omega(g(n)) = \{h(n) : \exists$ positive constant c, n_0 such that for any $c > 0, n_0 > 0$, and $0 \leq cg(n) < h(n)$, $\forall\ n \geq n_0\}$.

→ This is unlike the Ω notation where we say for some $c > 0$ (not any).

→ for ex, $5n^3 = \Omega(n^3)$ is asymptotically tight upperbound but $5n^2 = \omega(n^3)$ is non-asymptotically tight bound for $f(n)$.

Types of Time Function

(a) $O(1)$ = constant

if time complexity of any $f(n)=2$ than $O(1)$

$f(n)=5$ than $O(1)$

$f(n)=5000$ " "

because for each function $f(n)$ time complexity does not depend upon the size of input that is on "n".

(b) $O(\log n)$ → Logarithmic

(c) $O(n)$ - Linear → ex $f(n)=2n+3 = O(n)$

$f(n)=500n+700 = O(n)$

$f(n)=\frac{n}{5000}+6 = O(n)$.

(d) $O(n^2)$ - Quadratic →

(e) $O(n^3)$ = Cubic →

(f) $O(2^n)$ = exponential →

if may 2^n or, 3^n or n^n.

Compare classes of function

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

Why is it called Asymptotic Analysis?

In every case for a given function $f(n)$ we are trying to find another function $g(n)$ which approximate $f(n)$ at higher values of n. That means $g(n)$ is also a curve which approximates $f(n)$ at higher values of n.

In mathematics we call such a curve an "ASYMPTOTIC CURVE". In other terms, $g(n)$ is the asymptotic curve for $f(n)$. For this reason, we call algorithm analysis asymptotic analysis.

Simplyfying Properties of Asymptotic Notations

- Transitivity: $f(n) = \theta(g(n))$ and $g(n) = \theta(h(n)) \Rightarrow f(n) = \theta(h(n))$. Valid for O and Ω as well.
- Reflexivity: $f(n) = \theta(f(n))$. valid for O and Ω.
- Symmetry: $f(n) = \theta(g(n))$ if and only if $g(n) = \theta(f(n))$
- Transpose symmetry: $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$.
- If $f(n)$ is in $O(kg(n))$ for any constant $K > 0$, then $f(n)$ is in $O(g(n))$.
- If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$, then $(f_1 + f_2)(n)$ is in $O(\max(g_1(n), g_2(n)))$.
- If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$ then $f_1(n)\, f_2(n)$ is in $O(g_1(n)\, g_2(n))$.