Unit - 1

Algorithm :-

The word algorithm comes from the name of a Persian author, Abu Jafar Muhammad ibn Musa Al-Khwarizmi. An "algorithm" refers to a method that can be used by a computer for the solution of a problem.

⇒ An algorithm is a set of rules for carrying out calculation either by hand or on a machine.

⇒ An algorithm is a finite step-by-step procedure to achieve the required result.

⇒ An algorithm is a well-defined computational procedure that takes some values as input and produces some values as output. Thus an algorithm is a sequence of computational steps that transform the input into the output.

* An Algorithm is a finite set of instructions which, if followed accomplishes a particular task.

Every algorithm must satisfy the following :-
Characteristics of an Algorithm:

Input :- Zero or more quantities that are externally supplied.

Output :- At least one quantity is produced.

Definiteness :- Each instruction should be clear and unambiguous.

Finiteness :- Algorithm terminates after finite number of steps for all test cases.

Effectiveness :- Steps must be sufficiently simple and basic.

PSEUDOCODE

Pseudocode is a generic way of describing an algorithm without use of any specific programming language syntax. It is, as the name suggests, pseudo code it cannot be executed on a real computer, but it models and resembles real Programming code.

⇒ Pseudo code consists of short, English phrases used to explain specific tasks within a program's algorithm and there is no real formatting or Syntax rule.

⇒ It is an efficient and environment independent description of the key principles of an algorithm.

⇒ The purpose of using Pseudocode is that it is easier for humans to understand than conventional Programming language code.

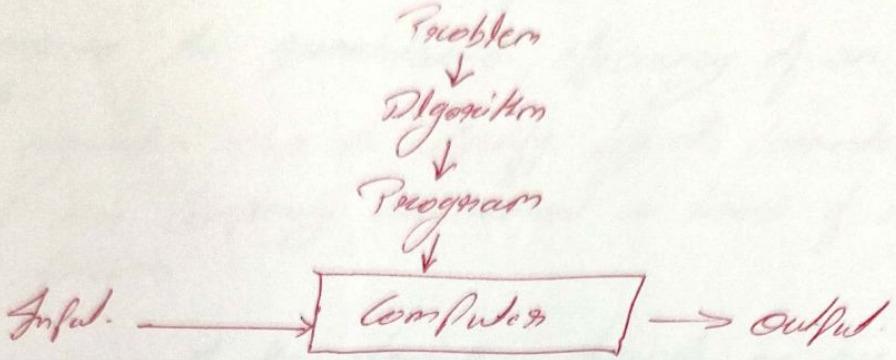
⇒ Pseudocode cannot be compiled or executed.

Program:

A program is a language specific implementation of the algorithm. A program is synonymous with code.

A computer program is another pervasive example of an algorithm. Every computer program is simply a series of instructions in a specific order, designed to perform a specific task. A solution for a given problem may be in the form of an algorithm or a program. An algorithm when

Expressed in some programming language is called a Program.



Analysis of Algorithm :

Analysis of an algorithm is required to dictate the correctness and measure the quantitative efficiency of an Algorithm. It includes.

- ⇒ Tracing of algorithm steps to specify logical correctness.
- ⇒ Space and time complexity measurement in terms of asymptotic order of growth of functions.
- ⇒ The performance of the algorithm depends on two main factors, the amount of computer memory consumed and the time required for successful execution of the algorithm.
- ⇒ There are two ways to analyze the performance of an algorithm first method is to carry out experiments using the various data sets after implementation of algorithm in any programming language and recording the amount of memory consumed and time required to execute it.
  Another method used before implementation is called analytical method in which one can approximately find out the space and time required by the algorithm.

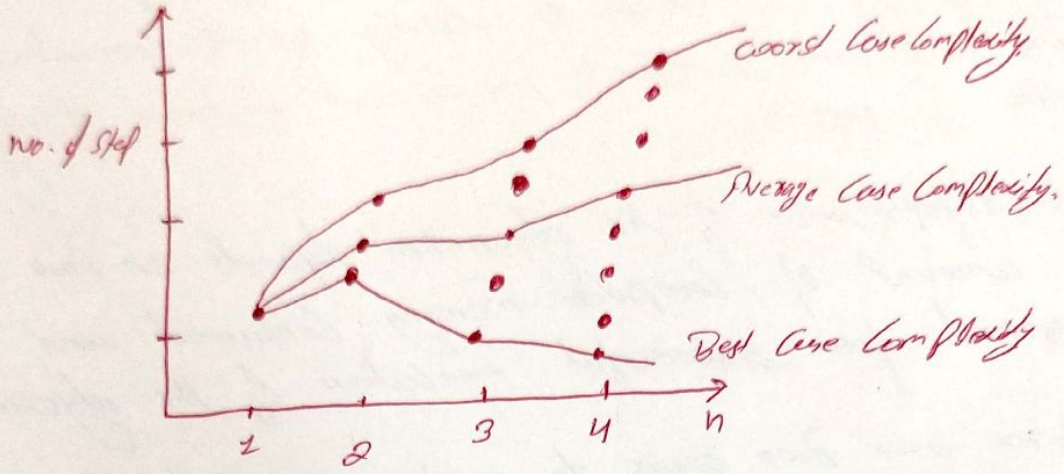
Cases Considered during the Analysis of an Algorithm

During the analysis of an algorithm, there are three different cases that are to be considered.

- Best case
- Average case
- Worst case

⇒ **Worst Case Complexity :-** The worst case complexity of the algorithm is the function defined by the maximum number of steps taken on any instance of size n. This represents the

input set that allows an algorithm to perform most slowly. for e.g. for a searching algorithm, the worst case is one where the value is in the last place or is not in the list.



Best Case Complexity :-

The best case complexity of the algorithm is the function defined by the minimum numbers of steps taken on any instance of size n. This represents the input set that allows an algorithm to perform most quickly.

- With this input the algorithm takes shortest time to execute as it causes the algorithms to do the least amount of work.

Average Case Complexity :-

The average case complexity of the algorithm is the function defined by an average number of steps taken on any instance of size n.

Complexity of Algorithm :-

Analyzing the complexity of an algorithm means establishing the amount of computing resources needed to execute the algorithm.
There are two types of resources.

→ Memory Space :-
It means the amount of space used to store all data processed by the algorithm.

→ Running time :-
It means the time needed to execute all the operations specified in the algorithm.

- Such an analysis is useful to establish if a given algorithm uses a reasonable amount of resources to solve a problem.
- The efficiency analysis (also called complexity analysis) is mainly used to compare algorithm in order to establish which is more efficient (that which uses less computing resources.)

Time Complexity :-

The analysis of time efficiency is based on estimating the time needed to execute the algorithm. The running time is not really the physical time corresponding to the execution of the algorithm on a computer but an estimate of the number of basic operations executed by the algorithm.

The running time of a statement = the cost of statement executing once * No. of times a statement is executed.

(8)

Space Complexity :-

Space Complexity of a program is the amount of memory required to execute a program successfully. This estimation of space can be used to identify the amount of memory required by the implementation.

The space complexity depends on two factors: Static allocation and dynamic allocation of memory.

In static allocation, the memory required by the program is known at compile time while in dynamic allocation it can be increase during execution.

We can divide the space needed by a program into two parts :-

a) A static part that includes the space required to store the code. The space required to store the code is compiler & machine dependent. The components constants, variables, complex data type are of this type.

b) A dynamic part, which consists of components whose memory requirement, depends on the instance of the problem being solved. Dynamic memory allocations and recursion are form composed of this type. The factors which can help us in determining the size of the problem instance are number of inputs, outputs etc.

GROWTH OF FUNCTIONS

Growth Rate of functions :

Resources for an algorithm are usually expressed as a function of input. often this function is messy and difficult to work. To study function growth easily; we reduce the function down to the important part.

$$f(n) = an^2 + bn + c.$$

Here n^2 term dominates the function, that is when n gets sufficiently large.

Dominates terms are what we are interested in to reduce a function, in which we ignore all constants and coefficients and look at the highest order term in relation to n.

⇒ Analysing Algorithm Control Statement :-

The analysis of an algorithm is calculated by considering its individual instructions. the individual instructions are calculated and then according to the control structure, we combine these lines.

Some algorithm of control structure.

1) Sequencing
2) If - then - else.
3) 'for' loop
4) while loop.
5) Recursion.

Orders of Growth :-

⇒ To analyze the efficiency of an algorithm we are interested in analyzing how the running time increases when the input size increases, but the detailed analysis of the running time of algorithm is not necessary. when two algorithms are compared with respect to their behaviours for the large input sizes, a useful measure is called order of growth.

⇒ the order of growth can be estimated by taking the dominant term of running time of the algorithm. Here we only specify how the running time increases as the input increases rather than specifying the exact relation between an algorithm's input and its running time for example, if the running time for an algorithm is an^2 with input size n, then we say that its running time scales as n^2 times when we increase the input size n.

Worst-Case, best-Case and Average-Case running time.

worst Case running time :-

⇒ the goodness of an algorithm is most often expressed in terms of its worst-case running time.

⇒ the worst-case efficiency of an algorithm is its efficiency for the worst-case input of size n, which is an input (or inputs) of size n for which the algorithm runs the longest among all possible inputs of that size.

⇒ The running time T(n) in the worst case provides very important information about the algorithm because it

binds its running time from the above (an upper bound).

⇒ For a simple linear search, in the worst case there are no matching elements or the first matching elements is in the last position in the list.

⇒ In this case the algorithm makes the largest number of key comparisons for searching an element in the array.

Best case running time :-

⇒ The best case running time is the minimum amount of time that an algorithm requires for an input of size n. It is the function defined by minimum number of steps taken on any instances of size n.

⇒ The term best-case performance is used in computer science to describe the way an algorithm behaves under optimal conditions. For Eg:- The best case for a simple linear search on a list occurs when the desired element is the first element of the list.

⇒ The best-case efficiency of an algorithm is its efficiency for the best-case input of size n, which is an input (or inputs) of size n for which the algorithm runs the fastest among all the inputs of that size.

⇒ The analysis of the best case efficiency is not nearly as important as that of the worst-case efficiency. But it is not completely useless.

Average Case running time

⇒ The average case time complexity is the function defined by the average number of steps taken on any instance of size.

⇒ Neither the worst-case nor the best-case can answer the question about the running time of a typical input or a random input. This is given by the average case efficiency of an algorithm.

⇒ Average-case running times are calculated by first arriving at an understanding of the average nature of the input, and then performing a running time analysis of the algorithm for this configuration.

Asymptotic Notation

Asymptotic means a line that tends to converge to a curve, which may or may not eventually touch the curve. It is a line that stays within bounds. The notation we use to describe the asymptotic running time of an algorithm are defined in terms of functions whose domains are the set of natural numbers $N = \{0, 1, 2, \ldots\}$. Such notations are convenient for describing the worst-case running time function $T(n)$, which is usually defined only on integer input sizes.

Big-oh Notations :-

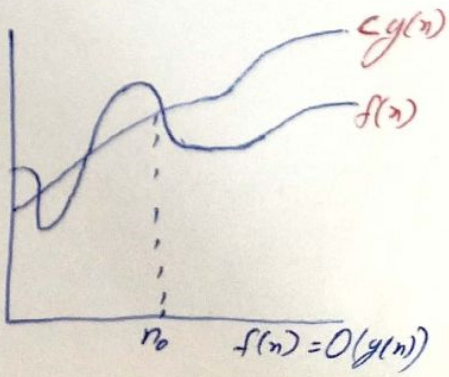
Big-oh is the formal method of expressing the upper bound of an algorithm's running time. It is the measure of the longest amount of time it could possibly take for the algorithm to complete.

Formally this means that if $g(n) \in O(f)$, $g(n) = cf(n)$ for all $n = n_0$ (where c is a positive constant) for non-negative functions, $f(n)$ and $g(n)$ if there exists an integer n_0 and a constant $c > 0$ such that all integers $n > n_0$

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

then $f(n)$ is Big-oh of $g(n)$. This is denoted as "$f(n) \in O(g(n))$"

i.e. the set of function which as n gets large, grow no faster than a constant times $f(n)$



Big- Omega Notation

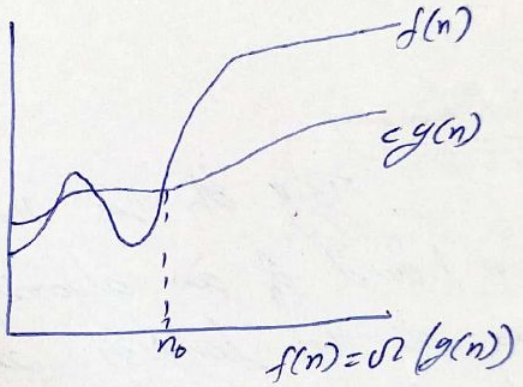
Just as O-notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound.

For non-negative functions, $f(n)$ and $g(n)$ if there exists an integer n_0 and a constant $c > 0$ such that for all integers $n > n_0$ $f(n) \geq cg(n)$

then $f(n)$ is big omega of $g(n)$. This is denoted as

"$f(n) \in \Omega(g(n))$"

& describes the best that can happen for a given data size.



Theta Notation

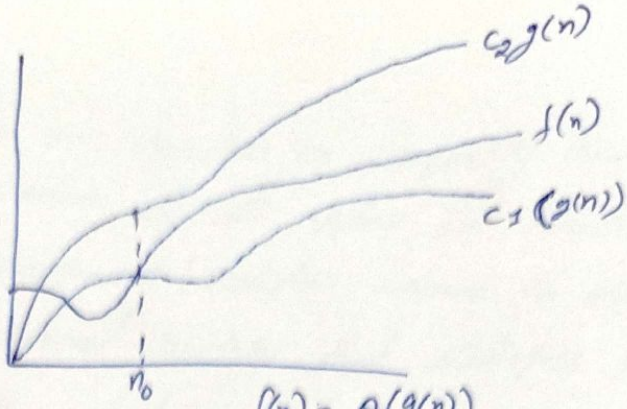
The lower and upper bound for the function T is provided by the theta notation for a given function $g(n)$.

For non-negative functions $f(n)$ and $g(n)$, if there exist an integer n_0 and positive constant c_1 and c_2 i.e., $c_1 > 0$ and $c_2 > 0$ such that for all integers $n > n_0$.

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

then $f(n)$ is theta of $g(n)$.

This is denoted as "$f(n) \in \Theta(g)$" can mean "f is order g"



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

Little - oh notation (o).

Asymptotic upper bound provided by O-notation may not be asymptotically tight. So o-notation is used to denote an upper bound that is asymptotically tight.

$o(g(n)) = \{f(n)$: for any +ve constant $c > 0$, if a constant $n_0 > 0$ such that

$0 \leq f(n) < c\,g(n) \;\forall\; n \geq n_0\}$

The function $f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity

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

for eg:- $2n = o(n^2)$

$2n^2 \neq o(n^2)$

Little Omega Notations :-

As o-notation is to O-notation, we have ω-notation as Ω-notation. Little - Omega (ω) is used to denote an lower bound that is asymptotically tight.

$\omega(g(n)) = \{f(n)$: for any +ve constant $c > 0$, if a constant $n_0 > 0$ such that.

$0 \leq c\,g(n) < f(n)$

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

Recurrence

A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs. To solve a recurrence relation means to obtain a function defined on the natural numbers that satisfies the recurrence. When an algorithm contains a recursive call to itself its running time can be described by a recurrence or recurrence equation. A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs.

For E.g.

The worst case running time $T(n)$ of the merge-sort procedure is described by the recurrence.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{if } n > 1 \end{cases}$$

For divide and conquer algorithms. We get recurrence like

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \\ aT\left(\frac{n}{b}\right) + D(n) + C(n) & \text{otherwise.} \end{cases}$$

a = number of subproblems

n/b = size of the subproblems (in terms of n)

$D(n)$ = time of divide problem into subproblems of size n.

$C(n)$ = time of combine the subproblem solutions to get the solution for the problem of size n.

(19)

There are three methods for solving this

Substitution method:- We guess a bound and then use mathematical induction to prove our guess correct.

Iteration method:- It converts the recurrence into a summation and then relies on techniques for bounding summations to solve the recurrence.

Master method:- It provides bounds for recurrence of the form.

$T(n) = aT\left(\frac{n}{b}\right) + f(n)$ where $a \geq 1$, $b > 1$ and $f(n)$ is a given function.

Substitution Method:-

The substitution method consists of two steps.

Step 1:- Guess the form of the solution.
Step 2:- Use mathematical induction to find the constants (boundary conditions) and show that the guess is correct.

guess the answer, and then prove it correct by induction.

The substitution method can be used to prove both upper bounds $O(n)$ and lower bounds $\Omega(n)$. When applying this method a good guess is vital. If the initial guess is wrong, the guess needs to be adjusted later.

Eg:- Consider the recurrence $T(n) = T\left(\lfloor \frac{n}{2} \rfloor\right) + 1$.
We have to show that it is asymptotically bound by $O(\log n)$

Sol

For $T(n) = O(\log n)$
We have to show that for some constant c.

$$T(n) \leq c \log n.$$

Put this in the given recurrence eqn.

$$T(n) \leq c \log\left(\lfloor \frac{n}{2} \rfloor\right) + 1$$
$$\leq c \log\left(\frac{n}{2}\right) + 1$$
$$= c \log n - c \log_2 2 + 1$$
$$\leq c \log n \text{ for } c \geq 1$$

Thus $T(n) = O \log(n)$

Eg:- Consider the recurrence $T(n) = 2T\left(\lfloor \frac{n}{2} \rfloor + 16\right) + n$.
We have to show that it is asymptotically bound by $O(n \log n)$

Sol

For $T(n) = O(n \log n)$ we have to show that for some constant c.

$$T(n) \leq cn \log n.$$

Put this in the given recurrence equation.

$$T(n) \leq 2\left[c\left(\lfloor \frac{n}{2} \rfloor + 16\right) \log\left(\lfloor \frac{n}{2} \rfloor + 16\right)\right] + n$$
$$= cn \log\left(\frac{n}{2}\right) + 32 + n$$
$$= cn \log n - cn \log 2 + 32 + n$$
$$= cn \log n - cn + 32 + n$$
$$= cn \log n - (c-1)n + 32$$
$$\leq cn \log n \text{ (for } c \geq 1)$$

Thus $T(n) = O(n \log n)$

(21) Sub

Eg:- Solve the recurrence $T(n) = 2T(n/2) + n$ using Substitution method.

Sol :- $T(n) = 2T(\lfloor n/2 \rfloor) + n$.

1. Guess :- $T(n) = O(n \log n)$ i.e., $T(n) \le cn \log n$ for some constant c.

2. Induction :- Base case :- we need to show that our guess holds for some base case (not necessarily $n = 1$, some small n is ok)

our guess does not hold for $n=1$, because the bound $T(n) = cn \log n$ yields

$$T(1) = c1 \log 1 = 0$$

which is contradiction coz $T(1) = 1$ now for $n=2$ & $n=3$.

$T(2) = 2$ $T(3) = 3$ $T(3) = 3 \log 3 = 4$.

are derived from the recurrence relation.

for $n=2$, $T(2) = 2T(1) + 2 = 2$.

$n=3$, $T(3) = 2T(1) + 3 = 3$.

So $T(2) \le c(2 \log 2)$ & $T(3) \le c(3 \log 3)$

for any $c \ge 2$

Inductive Step :-

Assume holds for $n/2$: $T(n/2) \le cn/2 \log n/2$

Now prove that it is holds for n: $T(n) \le cn \log n$

$T(n) \le 2T(\lfloor n/2 \rfloor) + n$ $T(1) = 1$.

$T(n) \le 2(c \cdot \lfloor n/2 \rfloor \cdot \log(\lfloor n/2 \rfloor)) + n$.

$\le c \cdot n \cdot \log(\lfloor n/2 \rfloor) + n$.

$\le c \cdot n \cdot \log n - c \cdot n \cdot \log 2 + n$

$\le c \cdot n \cdot \log n - c \cdot n + n$

$\le c \cdot n \cdot \log n$.

which holds for $c \ge 1$