Unit-1 :- Introduction to Algorithms

Algorithms, Pseudocode for expressing algorithms, Performance Analysis- Space complexity, Time Complexity, Asymptotic Notation- Big oh, Omega, Theta notation and Little oh notation, Polynomial Vs Exponential Algorithms, Average, Best and Worst case Complexities, Analysing Recursive Programs.

1) Algorithm :-

The word algorithm comes from the name of an parsian author Abu Jafar Mohammed ibn Musa-al Khowarizmi (c. 825 AD), who wrote a text book on mathematics where algorithm has come to refer to a method that can be used by a computer for the solution of a problem.

An algorithm is a finite set of instructions that accomplish a particular task. All the algorithms must satisfy these criterias.

(i) Input (ii) Output (iii) Definiteness
(iv) Finiteness (v) Effectiveness

Input :- zero (or) more quantities are externally supplied.

Output :- At least one quantity is produced.

Definiteness :- Each instruction is clear and unambiguous.

Finiteness :- If we trace out the instructions of an algorithm then the algorithm terminates after a finite number of steps.

Effectiveness:- Every instruction must be very basic, so that it can be carried out by a person to another person.

(4) Pseudocode Conventions:-

We can describe an algorithm in many ways. We can use a natural language like English. If we select this option we must ~~to~~ make sure that the resulting instructions are definite.

Most of the algorithms using a pseudocode that reassembles C language.

(i) Comments begins with // and continue untill the end of line.

(ii) Blocks are indicated with braces '{' and '}'.
A compound statement can we represented as a block. Statements are delimited by ";".

(iii) An identifier begins with a letter. The data type of variable are not explicitly declared. The type will be clear from the context. Weather a variable is global (or) local to a procedure will also evident from the context.

Compound data types can be formed with records.

```
node = record
{
    data type_1   data_1;
        :
        :
    data type_n   data_n;
        node      *link;
}
```

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

Variable := expression ;

(v) There are two boolean values TRUE and FALSE.

In order to produce these values the logical operators AND, OR and NOT and the relational operators <, ≤, >, ≥ ; ≠ are provided.

(vi) Elements of multi dimensional array are accessed using "[" and "]".

Ex:- If 'A' is a 2-D array then A[i, j].

(vii) The following looping statements are employed. i.e, for, while and repeat_untill (do while).

The while loop takes the following form

```
while (condition) do
  {
    statement-1 ;
        :
    statement-n ;
  }
```

As condition is true the statements get executed. When the condition becomes false the loop is exited. The value of condition is evaluated at the top of the loop.

The general form of for loop is

```
for variable := value 1 to value 2 step step do
  {
    stat-1 ;
      :
  } stat-n ;
```

Here values and step are arithmetic expressions. A variable of type integer are real are a numerical constant is a simple form of an arithmetic expression. The clause "step step" is an optional and takes as +1. step could either +ve (or -ve.

A repeat-untill (do while) statement is constructed as follows.

```
repeat
  {
    state-1;
      :
    state-n;
  } untill (condition)
```

Here the statements are executed as long as condition is failed. The value of condition is computed after executing the statement. The instruction break can be used within any of the above looping instructions to force exit. In The case of nested loop break result in the exit of the innermost loop.

A return statement within any of the above also will result in exiting the loop. A return statement result in the exit of the function it self.

(viii) The conditional statement has the following form

if condition then statement;

if condition then statement-1 else statement-2;

Here condition is a boolean expression and statements are orbitary statements. We also have the following case statement.

case
{ condition-1 : (state-1)
⋮
condition-n : (state-n)
else : (state-n+1)
}

Here statements are simple statements or compound statements. In case statements, if codition-1 is TRUE state-1 get executed and the case statement is exited. If stat-1 is FALSE condition-2 is evaluated. If none of the conditions are TRUE then statement-(n+1) is executed and case statement get exited.

(ix) Inputs and o/ps are done using the instruction read and write.

(x) There is only one type of procedure ("Algorithm") An algorithm consist of a heading and a body. The heading takes the form.

Algorithm Name (parameter list)

Where Name is the name of the procedure and parameter list is a listing of the parameter. The body has one or more statements enclosed within braces "{", "}".

Ex:- 1. Algorithm Max (A,n)
2. // "A" is an array of size n.
3. {
4. Result := a[i];

5. for i := 2 to n do
6. if a[i] > Result then := a[i];
7. return Result;
8. }.

③ Recursive Algorithm:-

A recursive function is a function that is defined in terms of it self. similarly an algorithm is said to be recursive, if the same algorithm is invoked in the body of the algorithm.

An algorithm that calls it self is a direct recursive.

An algorithm is said to be indirect recursive, if it calls by another algorithm.

Exf. Towers of Hanoi.

The towers of hanoi puzzle is fashinated after ancient towers of Brahma ritual. According to this there is a diamond tower labeled as 'A' with 64 golden disk. The disk were of decreasing size and were stacked on the tower in decreasing order of size from bottom to top.

Beside this tower there were two other diamond towers labeled as B and C.

Now, attempt, to move the disk from tower A to tower-B using tower-C as intermediate.

As the disk are very heavy they can be moved only one at a time. In addition to this at end time smaller disk-B on top.

For the given problem solution can be get by the age of recursion. Assume that the number of disks is 'n'. To get the largest disk to the bottom of the tower-B. We move the remaining (n-1) disk to tower-C and then move the largest disk to tower-B.

Now we are left with the track of moving the disk from tower-C to tower-B. To do this we have tower-A and tower-B. This can be performed with the help of recursion. The recursive nature of the solution for the towers of hanoi.

(i) Algorithm Towers of Hanoi (n, A, B, C).
(ii) // Move the top n disks from tower-A to tower-B.
(iii) {
(iv) if (n ≥ 1) then
(v) {
(vi) TowersofHanoi (n-1, A, C, B);
(vii) write ("move top disk from tower", A, "to top of tower", B)
(viii) Towers of Hanoi (n-1, C, B, A);
(ix) }
(x) }

(4) <u>Performance Analysis</u>:-

Performance evaluation can be divided into 2 major phases.

(i) Prior Estimation.
(ii) Posterior estimation.

To calculate the performance of any algorithm, we are going to be perform two estimations.

i.e, (i) Space Complexity and
(ii) Time Complexity.

Space Complexity:-

The space complexity of an algorithm is the amount of memory, it need to run to completion.
The space need by each algorithm is the sum of two factors.
i.e, Fixed part and linear part.

Constant Space Complexity and Linear Space Complexity:-

(i) A fixed part is independent of the characteristics of the i/p and o/p. This part typically includes the instruction space, space for simple variable and fixed size component variable, space for constants and so on

For example, (i) Algorithm sum (int a)
(ii) {
(iii) return a*a*100
(iv) }

(ii) A variable part that consist of space. The space needed by the component variable whose size is dependent on the particular problem instance being solved, The space needed by the reference variables, and the recursion stack space.
(iii) The space requirement $s(p)$ of any algorithm 'p' be written as $s(p) = c + S_p$

where, 'c' is a constant and S_p is a instance characteristics, which is used to measure the space requirement.

For example, Algorithm sum (A,n)

```
{
    sum = 0,i;
    for (i = 0; i<n; i++)
    {
        sum : = sum + A[n]
    }
    return sum;
}
```

The space needed for this algorithm is,

size variable 'n' = 1 word

Array value = n word

loop variable i = 1 word.

sum variable = 1 word.

Space requirement $S(P) = 1 + (n+1) + 1$

$= 3 + n$

Another example, Algorithm Rsum (A,n)

```
{
    if (n ≤ 0) then
        return 0.0;
    else
        return RSum (A, n-1) + A[n];
}
```

The recursive stack space includes space for formal parameters, the local variables and the return address.

Return address requires only one word of memory.

Each call to Rsum requires at least 3 words. Since, the depth of recursion is n+1. Then the recursion stack space needed is

$$S(p) = 3(1+n).$$

Time Complexity:-

The time complexity of an algorithm is the amount of computer time, it need to run to completion. The time T(P) taken by the program 'p' is the sum of compilation time and run time (Execution time). The compilation time does not depend on the instance characteristics.

A compiled program will be run several times without recompilation. So we consion only the runtime of a program. This run time is denoted by t_p [Instance Characteristics].

The value of $t_p(n)$ for any given 'n' can be obtained only experimentally. To solve a problem instance with characteristics given by 'n', we might block all other operations and obtain a count for the total number of operations. So, we can count only the number of program step.

A program step is defined as a syntactically or simantically meanigful segment of a program. That has an execution time that is independent of instance characteristics.

The number of steps any program statement is assigned depends on the kind of statements.

Comments = 0 steps

Assignment statement = 1 step

Conditional Statement = 1 step

Loop Conditions for in times = 1+n steps

Body of loop = n steps.

We can determine the number of steps needed by a program to solve a particular problem instance in 2 ways. In the first method, we introduce a new variable (count) in to the program this is a global variable with initial value zero. Statements to increment count by the appropriate amount are introduced in to the program. So 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= 0; i<n; i++)
    {
        count := count +1; // For "for".
        s:= s+ a[i];
        count := count +1; // For assignment
    }
```

count := count + 1; // For last time of for
count := count + 1; // For the return.
return s;
}

The value of count in the for loop will be increases by 2n and the total number of steps program steps for algorithm is 2n+3.
The step count is useful to represent the runtime for a program changes with changes in the instance characteristics from the step count for algorithm sum, we see that if n is double the run time also doubles so the runtime grows linearly in n.

The second method to determine the step count of an algorithm is to build a table in which we list the total number of steps contributed by each statement.

In table first determine the number of steps for execution (s/e) of statement and the total numbers of times (frequency) each statement is executed.

The s/e of a statements is the amount by which the count changes as a result of the execution of that statement by combining this 2 quantities the total contribution of each statement is obtained by adding the contribution of all statements the step count for the entire algorithm is obtained.

Ex:- 1:-

statement	s/e	frequency	Total steps
Algorithm sum(A,n)	0	—	0
{	0	—	0
s:=0;	1	1	1
for (i=0; i<n; i++)	1	1+n	1+n
s:=s+A[i];	1	n	n
returns;	1	1	1
}	0	—	0
Total.			2n+3.

Ex-2:- Recursive Algorithm

Statement	s/e	frequency n=0 I	frequency n>0 F	total steps n=0 T	total steps n>0 F
Algorithm (A,n)	0			0	0
{	0	—	—	0	0
If (n≤0) Then	1	1	1	1	1
return 0.0;	1	1	0	1	0
else					
return	1+x	0	1	0	1+x
Rsum(A,n-1)+R(n)				0	
}	0	—			
Total				2	x+2

Ex-3 :- Inner loops.

Statements	s/e	frequency	Total steps
Algorithm add(A,B(m,n))	0	—	0
{	0	—	0
for (i=0; i<m; i++)	1	1+m	1+m
for(j=0; j<n; j++)	1	1+n	m+mn
c[i,j] = a[i,j]+B[i,j]	1	mn	mn
}	0	—	0
Total			2m+2mn+1.

(5) Asymptotic Analysis :-

Asymptotic Analysis of an algorithm refers to define the mathematical boundations (framings) of its run time performance using asymptotic analysis. We can very well conclude the best case, average case and worst case scenarios of an algorithm.

Asymptotic analysis is input bound, that is if there is no input to the algorithm then it is concluded to work in a constant time.

Asymptotic analysis refers to computing the running time of any operation in mathematical units of computations (calculations).

For example, the running time of one operation is computed as f(n) and may be for another operation. It is computed as $g(n^2)$. This means the first operation running time will increase linearly with the increase of in 'n' and the running time of second operation will increase exponentially when 'n' increases. ||ly, the running time of both operations will be nearly the same if 'n' is significantly small.

The time required by an algorithm falls under 3 types.

(i) Best Case
(ii) Avg Case
(iii) Worst Case

Best Case:- Minimum time required for the program execution.

Average Case:- Average time required for the program execution.

Worst Case:- Maximum time required for the program execution.

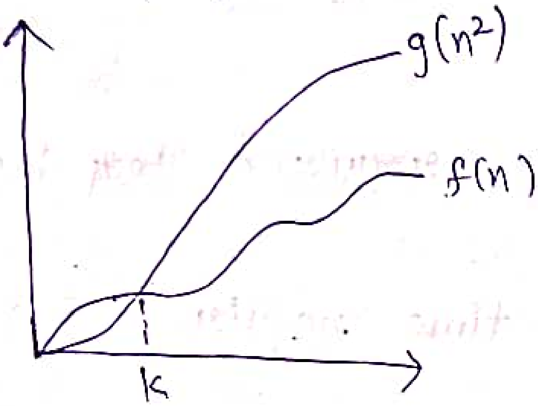
Asymptotic Notations:-

The commonly used asymptotic notations to calculate the running time complexity of an algorithm are divided into 5 types.

(i) Big - oh Notation (O - Notation)
(ii) Omega - Notation (Ω notation)
(iii) Theta Notation (θ - Notation)
(iv) Little oh Notation (o - Notation)
(v) Little Omega Notation (ω - notation).

Big-oh Notation:-

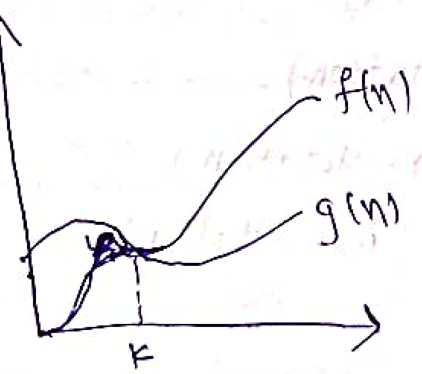
The notation O(n) is the formal way to express the upper bound of an algorithm running time. It means the worst case time complexity or longest amount of time an algorithm can possibly take to complete.



Here, the function $f(n) = O(g(n))$
There exist positive constant $c > 0$ and n_0 such that $f(n) \leq c \cdot g(n) \ \forall \ n > n_0$.

Omega Notation:-

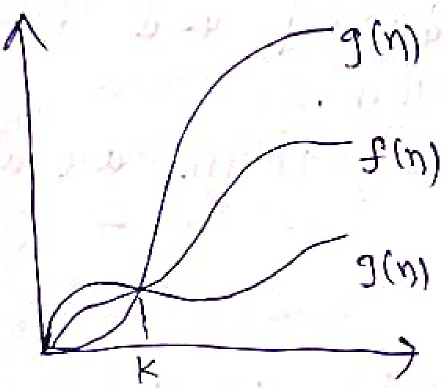
The notation $\Omega(n)$ is the formal way to express the lower bound of an algorithms running time. It measures the best time complexity or best amount of time an algorithm can possibly take to complete.



The function $f(n) = \Omega(g(n))$ and there exist +ve constant $c > 0$ and n_0 such that $g(n) \leq c \cdot f(n)$ for all $n > n_0$.

Theta Notation:-

The notation $\Theta(n)$ is the formal way to express both lower bound and upper bound of an algorithms running time and its represents as follows.



The function $f(n) = \Theta(g(n))$ and there exist the possible constant (C_1 and C_2) > 0 and n_0 such that

$$C_1 \cdot g(n) \leq f(n) \leq C_2 \cdot g(n) \quad \forall\ n \geq n_0.$$

Little oh Notation:-

Little oh notation is used to describe an upper bound that can not be tight in other words loosely upper bound of an algorithm.

Let the function $f(n)$ is $f(n) = o(g(n))$ and there exist possible constant $c > 0$ and there exist an integer constant that is $n_0 \leq 1$ such that $f(x) > 0$.

Little Omega Notation:-

The little omega notation is used to describe a loosely lower bound of $f(n)$. The function $f(n) = \omega(g(n))$ and there exist a possible constant $c > 0$ and there exist an integer constant that is $n_0 \leq 1$ such that $f(n) < 0$.

Polynominal Vs Exponential Algorithms:-

In general time complexities are classified as constant, linear, logarithmic, polynominal, exponential etc.

Among these the polynominal and exponential are the most pominently considered and defines the complexity of an algorithm.

These 2 parameters for any algorithm are always influenced by size of inputs.

Polynominal Running Time:-

An algorithm is said to be solvable in polynomial time if the number of steps required to complete the algorithm for a given input is $O(n^k)$.

For some positive integers k, where n is the complexity of the input. Polynominal time algorithm are said to be false.

Most familiar mathematical operations such as addition, substraction, multiplication and division, as well as computing square roots, powers and logarithms can be performed in polynominal times.

Exponential Running Time:-

The set of problems which can be solved by an exponential time algorithms, but for which no polynominal time algorithms is unknown. An algorithm is said to be exponential time, if $T(n)$ is upper bounded by $2^{poly(n)}$,

where, poly(n) is polynominal in n.

An algorithm is exponential time if T(n) is bounded by $O(2^{nk})$ for some time.
Algorithm which have exponential time complexity grows more faster than polynominal algorithm.
For example, polynominal time complexity $= n^3+2n^2+1$
and exponential equation $= 2^n$
if $n = 1000$, the compared with these two equations exponential equation get huge value.

Average, Best and Worst Case Complexity:-

Worst Case Analysis:- Big-O

In worst case analysis we calculate the upper bound on the running time of algorithm. We must know the case that uses a maximum number of operations can be executed. We define an algorithms time complexity in worst case by using Big-Oh notation, which determines the set of functions grows slower than (or at) the same rate as the expression.
For linear search the worst case happens, when the elements to be searched (x) is not present in the array. When the 'x' is not present the search function compares it with all the elements of array one by one. Therefore the worst case time complexity of the linear search is O(n).

Best Case Analysis:- Ω

In bestcase analysis, we calculate the lower bound on the running time of an algorithm. Here we must know the cases that causes a minimum number of operations, to be executed

We define an algorithms best case time complexity by using "Omega" notation.

which determines the set of functions will grows faster or at the same rate at the same expression. It explains the minimum amount of time an algorithm requires to consider all input values.

In the linear search problem the best case occurs when 'x' is present at the first location. So the number of operations in the best case is constant. Then the time complexity is $\Omega(1)$

Average Case Analysis:- Θ

In average case analysis we take all possible inputs and calculate the computing time for all of the inputs. the calculated values are then divided the sum by the total number of inputs. we define the algorithm's average case time complexity by using the 'Θ' notation. which defines the set of functions lies in both O(expression) and Ω(expression). This is how we define a time complexity for the average case algorithms

For the linear search problems, Let us assume that all cases are unformally distributed. So, we sum all the cases and divide the sum by (n+1) then the time complexity for the average case is

Average case Time $= \sum_{p=1}^{n} \frac{\theta(1)}{(n+1)} = \frac{\theta\left(\frac{(n+1)*(n+2)}{2}\right)}{(n+1)}$

$= \theta(n).$