Module - 1

Design & Analysis of Algo

Nithin Kumara
Asst prof
Dept of CSE
VVCE, Mysuru

* **Algorithm :-**

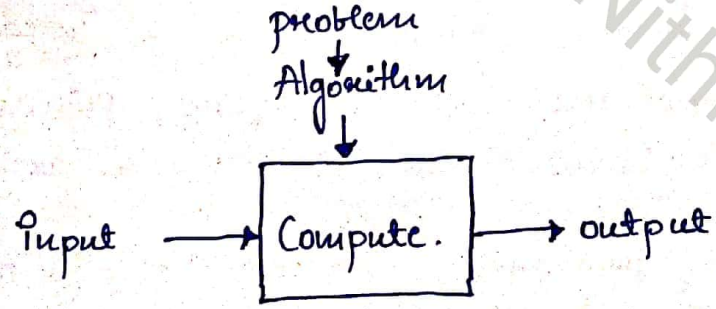
An Algorithm is an Effective, Efficient & Best Method which can be Used to Express Soln of Any problem within a finite Amount of Space & time in a well defined formal language.

"An Algorithm is defined as finite Sequence of Unambitiguous Instruc-tions followed to Accomplish particular task".

→ All Algorithms Must Satisfy the foll Criteria

* Input → One or More Quantity can be Supplied.
* output → Atleast One Quantity is produced.
* Definiteness → Each Instruction of the Algo Should be clear & Unambiguous
* Finiteness → The procesn Should be terminated After a finite no of Steps.
* Effectiveness → Every Instruction Must be basic Enough to be Carried out theoretically or by using "pencil" & "paper"

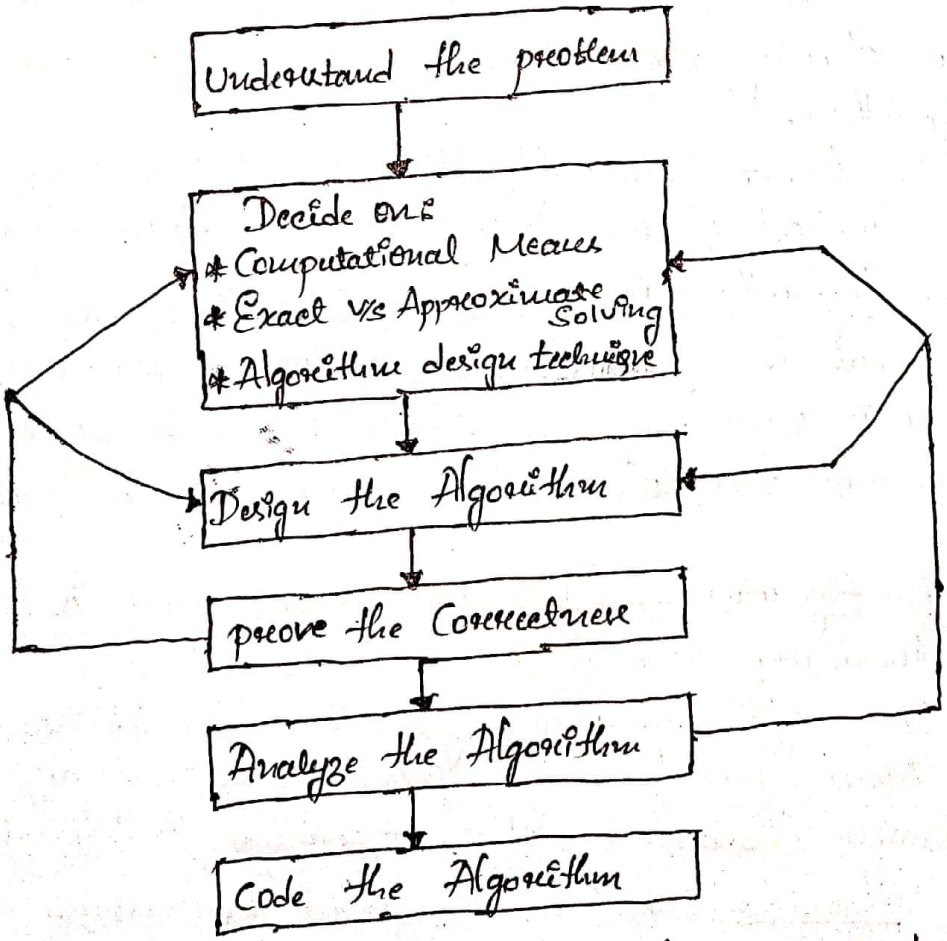
→ The Definition of Algorithm Can be depicted using the foll diagram



→ Different Approaches used to Solve the given problem using Algorithm

* Brute-Force Method (Straight-Forward)
* Divide-and-Conquer
* Decrease-and-Conquer
* Greedy Method
* Dynamic programming
* Transform-and-Conquer
* Backtracking
* Branch-and-Bound.

* Fundamentals of Algorithmic problem Solving:- A Sequence of Steps one typically goes through in designing & Analyzing an Algorithm.



* Understanding the problem → The first thing you need to do before designing an Algorithm is to understand Completely the given problem, The Input & Set of Instances the Algorithm needs to handle.

* Ascertaining the Capabilities of Computational device → Once you Completely understand a problem, you need to Consider the Capabilities of Computational device & Select Appropriate programming Model Such as "Sequential" or "parallel".

* Choosing between Exact & Approximate problem Solving → The next principal decision is to choose between Solving the problem "Exactly" or "Approximately". There are Important problems that Simply Cannot be Solved Exactly Such as Extracting Square roots, Solving Linear Equation, Etc.

* Algorithm design techniques → is a general Approach to Solving problems Algorithmically that is Applicable to a Variety of problems from different Areas of Computing. i.e problems for which there is no known Satisfactory Algorithm.

* Designing an Algorithm & Data Structures → One Should pay close Attention to Choosing data Structures for the operations performed by the Algorithm.
For Ex: the Linear Search Algorithm would run Longer if we use a LinkedList Instead of an Array in its Implementation.

* proving an Algorithm Correctness → Correct output for Every legitimate Input in finite time. For Some Algorithms, a proof of Correctness is Quite Easy, for others it can be Quite Complex. So the Best Common technique to prove Correctness is "Mathematical-Induction".

* Analyzing An Algorithm → After Correctness, most Important is Efficiency. There are two parameters to be Considered
  * Time Efficiency → How fast the Algorithm runs
  * Space Efficiency → How much Extra Memory it uses.

Another desirable characteristic of an Algorithm is "Simplicity".

* Coding an Algorithm → Most Algorithms are destined to be ultimately Implemented as Computer programs. Implementing an Algorithm Correctly is necessary but not Sufficient, Modern Compilers & powerful Languages Should be used with Code optimization mode.

* Analysis Framework :-

The Analysis of Algorithm means the Investigation of an Algorithm's Efficiency Interms of time required for its Execution & Extra Memory Space taken by it. But, now in new technological Era, as we have got Computers with huge Storage Space.

We won't bother much about Space Complexity in practical problems.

→ The foll are some important Aspects that we come across in Analysis Framework

* Measuring an Input's Size → It is obvious that all Algo take more time for Execution, if the Input is large.

→ For Ex Sorting a larger list, Multiplying Matrices of big orders Etc take much time

In the problems like Sorting, Searching Etc, the Array Size itself will decide the time taken for Execution.

→ The input Size is usually Measured in bits as

$$b = \lfloor \log_2 n \rfloor + 1$$

* Units for Measuring Running time → The Algorithm's Execution time can be Measured in Seconds, Milliseconds Etc This Execution time depends on

* Speed of a Computer
* Choice of the language to Implement Algo
* Compilers used for generating Code
* Number of Inputs

Depending on all such Aspects, it is quite difficult to find out the time required so, we will go for one Appreciable

& Believable Approach
Where the time required for Each of the Executable State-ment is Considered.
→ We will ignore Some non-Important Statements like Input & output
Concentrate only on Very Important operation called "Basic-Operation".
This will Contribute More to the total running time.

* Worst-Case, Best-Case & Average-Case Efficiencies :- The time Complexity of Some Algorithms depends on Size of Input. But, In Some other Cases, this is not true.
→ For Ex, In case of Searching Algo. The time depends on the position of Element to be Searched.
If the Element is found In first position of the list itself then time taken by Algo will be obviously less.
→ Thus, for the Same Input Size 'n', the time Complexity differs.
So, we will divide the Efficiency of an Algo into three Categories
* Worst-Case Efficiency
* Best-Case Efficiency
* Average-case Efficiency

* Worst-Case Efficiency :- The Efficiency of Algorithm for the Worst-Case Input of Size 'n' for which the Algorithm takes "longest time" to Execute is Called as "Worst-case Efficiency".
→ For Ex :- In linear Search Algo requires n Comparisons, key is

present in the last position

$$C_{worst}(n) = n$$

This indicates the time required for this Algorithm to run.

* Best-Case Efficiency :- The efficiency of an Algorithm for the input of size n for which Algorithm takes "Least-time" for Execution is Best-Case Efficiency.

→ For Ex, In Linear Search Algo when the Element is found at first position.

$$C_{Best}(n) = 1$$

* Average-Case Efficiency :- In real life Situation, we come Across worst-Case & Best-Case Very rarely.
Usually, the Elements of the list are Randomly distributed. so, we will go for Measuring Average time.

→ For Ex, The Average-Case Efficiency of Linear Search Algo is as shown below

$$C_{Avg}(n) = \frac{p(n+1)}{2}$$

* Order of Growth :-
Suppose that we have got two Algorithms for Solving the Same problem. If the Size of input is very Small then we can't Judge, which is better one.
we've to check the time taken by the Algorithm as the input Size increases this is known as "Order of Growth"

→ The Algorithm that we come across are having Execution-time proportional to any of the following functions.

* Constant → If Most of Instructions in a pgm are Executed only once or very few no of times, we will say that the Running time of a program is Constant.

* $\log n$ → If the difference b/w the Input n & running time such -case logarithmically as n increases then we say that running time is a function $\log n$.

* n → If the Execution time is 'n' for the Input n, then it indicates Algorithm is Linear.

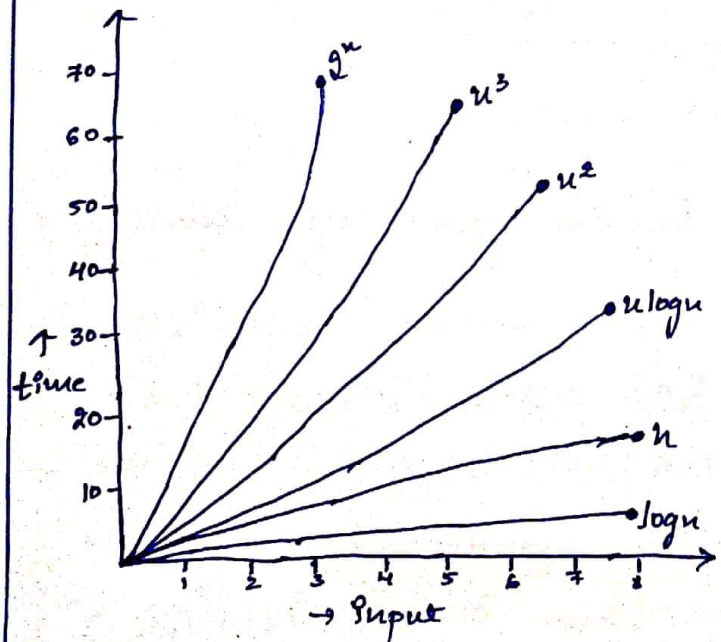
* $n\log n$ → the time taken by Algorithm for the Input n is $n\log n$.
This result is found in Algo that Solve the problem into no of Smaller Subproblems & then their Subproblems are Solved Individually & finally Combined to get final Soln.
Ex Mergesort & Quicksort

* n^2 → Indicates the running time of Algorithm is "Quadratic" This kind of Algo are used when 'n' is Relatively Small.

* n^3 → Indicates that, running time is "Cubic" & Applicable for Smaller problems.

* 2^n & $n!$ → This Indicates the Execution time is Exponential.

→ Consider the foll table & graph of order of Growth.



n	$\log n$	$n\log n$	n^2	n^3	2^n	$n!$
1	0	0	1	1	2	1
2	1	2	4	8	4	2
4	2	8	16	64	16	24
8	3	24	64	512	256	40320
16	4	64	256	4096	65536	High
32	5	160	1024	32768	High	very High

* Asymptotic Notations :-

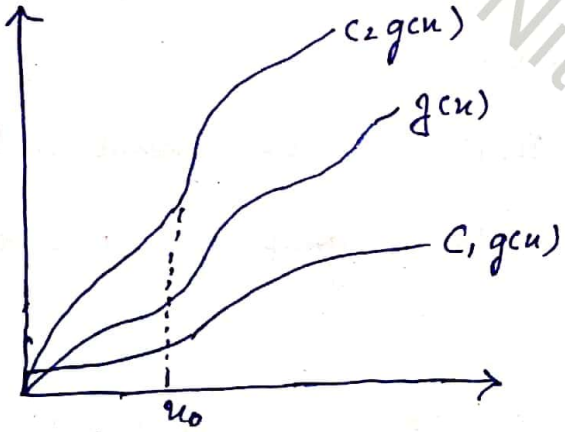
Are used to define the running time of an Algorithm.

We're Studying the "Asymptotic" Efficiency of Algorithms to know how the running time of an Algo increases with the Size of input in limit (Bound).

* Θ-notation → For a given function g(n), we denote by Θ(g(n)) the Set of functions.

∴ Θ(g(n)) = {f(n): there Exist positive Constants c_1, c_2 & n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0$}

→ A function f(n) belongs to the Set Θ(g(n)), if there Exists positive Constants c_1 & c_2 Such that it can be "Sandwiched" b/w $c_1 g(n)$ & $c_2 g(n)$ for Sufficiently large n.

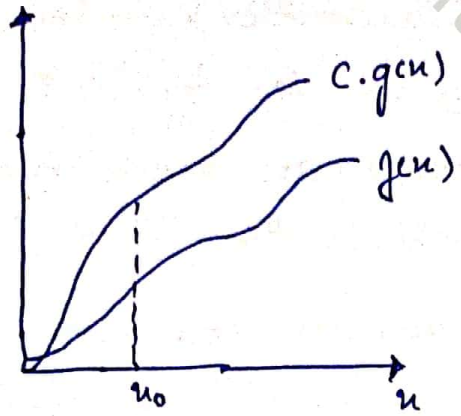


$f(n) = \Theta(g(n))$

* O-notation → For a given function g(n) we denote by O(g(n)) the Set of functions

∴ O(g(n)) = {f(n): there Exist positive Constant c & n_0 such that $0 \le f(n) \le c(g(n))$ for all $n \ge n_0$}

→ We use 'O' notation to give an upperbound an function. If f(n) is bounded above by Some Constant Multiple (c) of g(n) for all large n.

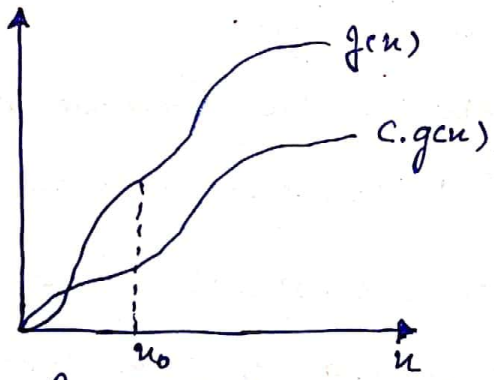


$f(n) = O(g(n))$

* Ω-notation → For a given function $g(n)$, we denote by $\Omega(g(n))$ the set of functions.

$\therefore \Omega(g(n)) = \{f(n)$: there Exist positive Constant c & n_0 Such that $0 \le cg(n) \le f(n)$ for all $n \ge n_0\}$

→ We use Ω notation to give an lowerbound an function. If $f(n)$ is bounded below by some Constant Multiple of $g(n)$ for all large n.



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

* little o-notation → The Asymptotic upperbound provided by O-notation May or May not be Asymptotically tight.

→ We use o-notation to denote an upper-bound that is not Asymptotically tight $o(g(n))$ is the set of functions.

$\therefore o(g(n)) = \{f(n)$: for any +ve constant $c > 0$ such that $0 \le f(n) < c\,g(n)$ for all $n > n_0\}$

* little ω- notation :- The Asymptotic lower bound provided by Ω - notation may or may not be Asymptotically tight.

→ We use ω-notation to denote an lower-bound that is not Asymptotically tight $\omega(g(n))$ is the set of functions.

$\therefore \omega(g(n)) = \{f(n)$: for any +ve constant $c>0$ such that $0 \le cg(n) < f(n)$ for all $n > n_0\}$

* property of Asymptotic Notation if an Algorithm has two Execut-able parts, the Analysis of this Algo can be obtained by the foll proof

* prove the foll theorem if $f_1(n) \in O(g_1(n))$ & $f_2(n) \in O(g_2(n))$ then $f_1(n) + f_2(n) \in O(\max\{g_1(n), g_2(n)\})$

⇒ By defn w.k.t $f(n)$ is said to be big-oh of $g(n)$ denoted by

$f(n) \in O(g(n))$

Such that there Exists a +ve Constant c & n_0

$f(n) \le c.g(n)$ for all $n \ge n_0$

it is given that $f_1(n) \in O(g_1(n))$, there Exists a relation

$f_1(n) \le c_1.g_1(n)$ for $n \ge n_1$ ------- (1)

it is given that $f_2(n) \in O(g_2(n))$, there Exists a relation

$f_2(n) \le c_2.g_2(n)$ for $n \ge n_2$ ------- (2)

let us Assume $c_3 = \max\{c_1, c_2\}$ & $n = \max\{n_1, n_2\}$ ----- (3)

By adding (1) & (2) we have

$f_1(n) + f_2(n) \le c_1 * g_1(n) + c_2 * g_2(n)$

$\le c_3 * g_1(n) + c_3 * g_2(n)$

$\le c_3 [g_1(n) + g_2(n)]$

$\le c_3 * 2 \max\{g_1(n), g_2(n)\}$

Since $f_1(n) + f_2(n) \leq c_3 * 2 \max\{g_1(n), g_2(n)\}$ By def^n we write

$$\boxed{f_1(n) + f_2(n) \in O(\max\{g_1(n), g_2(n)\})}$$ //

* Order of Growth using limits :- Asymptotic Notations such as Big-oh, Big-Omega, & Big-theta can be defined as

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \begin{cases} 0 \rightarrow f(n) \in O(g(n)) \\ c \rightarrow f(n) \in \Theta(g(n)) \\ \infty \rightarrow f(n) \in \Omega(g(n)) \end{cases}$$

Before we Compare the Order of Growth, Anyone Must remember all two formula's

* L-Hospital rule $\rightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$

* Sterling's rule $\rightarrow n! = \sqrt{2\pi n}\,(n/e)^n$

* Compare the order of Growth $\frac{1}{2}n(n-1)$ & n^2

$\Rightarrow f(n) = \frac{1}{2}n(n-1)$ $\quad g(n) = n^2$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \Rightarrow \lim_{n\to\infty} \frac{\frac{1}{2}n(n-1)}{n^2} = \frac{1}{2}\lim_{n\to\infty} \frac{n^2 - n}{n^2}$$

$$= \frac{1}{2}\lim_{n\to\infty}(1 - 1/n)$$

$$= \frac{1}{2}(1 - 1/\infty)$$

$$= \frac{1}{2} = \text{constant}$$ //

$$\therefore \frac{1}{2}n(n-1) \in \Theta(n^2)$$ //

* Compare the order of Growth of $\log_2 n$ & $\sqrt{n}$

$\Rightarrow f(n) = \log_2 n \quad g(n) = \sqrt{n}$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\log_2 n}{\sqrt{n}} = \lim_{n\to\infty} \frac{\left[\frac{\log_e n}{\log_e 2}\right]}{\sqrt{n}}$$

$$= \lim_{n\to\infty} \frac{1/\log_e 2 \ (\log_e^n)}{\sqrt{n}}$$

$$= \lim_{n\to\infty} \frac{\log_e 2 \ (\log_e^n)}{\sqrt{n}}$$

$$= \log_e 2 \left[\lim_{n\to\infty} \frac{\log_e^n}{\sqrt{n}}\right]$$

According to L-Hopital rule

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)} = \log_e 2 \left[\lim_{n\to\infty} \frac{(\log_e^n)'}{\sqrt{n}'}\right] \text{ ----- (1)}$$

* derivative of $\log_e^n = 1/n$ ------ (2)

* derivative of $\sqrt{n} = n^{1/2} = x^n = n.x^{n-1}$

$= 1/2 . n^{1/2-1}$

$= 1/2 \ n^{-1/2}$

$= 1/2n^{1/2} = 1/2\sqrt{n}$ ----- (3)

Substitute (2) & (3) in (1)

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \log_e 2 \left[\lim_{n\to\infty} \frac{1/n}{1/2\sqrt{n}}\right]$$

$$= \log_e 2 \left[\lim_{n\to\infty} \frac{2\sqrt{n}}{n}\right]$$

$$= 2\log_e 2 \left[\lim_{n\to\infty} \frac{\sqrt{n}}{n}\right] = 0 //$$

$\therefore \log_2^n \in O(\sqrt{n}) //$

* Compare the order of Growth b/w $n!$ & 2^n

$\Rightarrow f(n) = n! \quad g(n) = 2^n$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n!}{2^n} = \lim_{n\to\infty} \frac{\sqrt{2\pi n}\,(n/e)^n}{2^n}$$

$$= \lim_{n\to\infty} \sqrt{2\pi n} \left[\frac{n^n}{2^n e^n}\right]$$

$$= \lim_{n\to\infty} \sqrt{2\pi n} \left(\frac{n}{2e}\right)^n = \infty$$

$n! \in \Omega(2^n) //$

* Mathematical Analysis of Non-recursive Algorithms

The General plan for Analysis is as shown below

* Based on input size, decide the various parameters to be considered.
* Identify the "Basic operation" of Algorithm.
* Compute the no of times the Basic operation is Executed.
* Obtain the total no of times a Basic operation is Executed.
* Simplify using Standard formula & Compute the order of Growth.

→ In this part of Non-recursive Algo Analysis, you'll learn the following Algorithms.

* Maximum of n Elements
* Matrix Multiplication
* Uniqueness problem

→ The formulas used for Analysis are

1. $\sum_{i=1}^{u} c \cdot a_i = c \sum_{i=1}^{u} a_i$ //

2. $\sum_{i=1}^{u} (a_i \pm b_i) = \sum_{i=1}^{u} a_i \pm \sum_{i=1}^{u} b_i$ //

3. $\sum_{i=1}^{u} 1 = \text{Upper limit} - \text{Lower limit} + 1$ //

4. $\sum_{i=0}^{u} i = \sum_{i=1}^{u} 1 = 1 + 2 + \dots u = \frac{u(u+1)}{2}$ //

5. $\sum_{i=1}^{u} i^2 = 1^2 + 2^2 + \dots + u^2 = \frac{u(u+1)(2u+1)}{6} = \frac{1}{3}u^3$ //

* Finding the Largest Element in a list of n nos

* Algorithm MAX (A[0.... n-1])

I/p → An array of real nos A[0... n-1]

O/p → the Value of Largest Element in A

```
MAX ← A[0]
    for I ← 1 to n-1 do
        if A[i] > MAX
            MAX ← A[i]
    Return MAX
```

* Analysis :-

* The Algorithm depends on Input Size

* There are two operations
  * Assignment → happens only once
  * Comparison → happens for all value of n

* As loop is Executed n-1 times, the Comparison happens n-1 times

$$\therefore \text{no of times operation Executed} = \sum_{i=1}^{n-1} 1$$

$$T(n) = \sum_{i=1}^{n-1} 1$$

$$= (n-1-1+1) \quad \because \text{Upperlimit} - \text{lowerlimit} + 1$$

$$= n-1$$

$$T(n) = \Theta(n)$$

* Multiplication of two nxn Matrices 'A' & 'B'

* Algorithm MatrixMul (A[i,j], B[i,j])

I/P:- nxn Matrices A & B

O/P:- Result Matrix C = AB

```
for i ← 0 to n-1 do
    for j ← 0 to n-1 do
        C[i,j] ← 0
        for k ← 0 to n-1 do
            C[i,j] = C[i,j] + A[i,k] * B[k,j]
Return C
```

* Analysis :-
  * Input Size is n & it is the only parameter
  * There are two basic operations
    * Addition → happens Same no of times
    * Multiplication → More no of times (Basic operation)
  * The Basic operation depends on 'i'. It is done for Each value of k, i, & j so 3 for Statements can be written as

$$\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1}$$

$$\therefore T(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1$$

$$= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} [(n-1-0+1)] \quad \because \text{Upperlimit} - \text{lowerlimit} + 1$$

$$= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n$$

$$= n \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 1$$

$$= n \sum_{i=0}^{n-1} [(n-1-0+1)]$$

$$= n^2 \sum_{i=0}^{n-1} 1$$

$= n^2 [(n-1-0+1)]$

$= n^3$

$\therefore T(n) = \Theta(n^3)$ //

* Finding whether all the Elements of given Array are distin-ct or not (Uniqueness problem)

* Algorithm Unique A([0....n-1])

  I/p :- An array A[0....n-1]

  O/p :- True or FALSE

```
for i ← 0 to n-2 do
    for j ← i+1 to n-1 do
        if A[i] == A[j]
            return FALSE
return TRUE
```

* Analysis :-

* The parameter to be Considered here is Input Size 'n'

* The Basic operation is "Comparison".

* The Algo Stops, when Encounters duplicate in list, then Algo depends on 'n'

  But we can't given the position of duplicate, so we will Assume the Worstcase treating all Comparisons are Made.

* for Statements can be Written as

$$\sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1}$$

$$\therefore T(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

$$= \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1]$$

$$= \sum_{i=0}^{n-2} \{n-1-i\}$$

$= (n-1) + (n-2) + (n+3) \text{ ----- } 3+2+1+0$

$= 1+2+3+ \text{ ---- } (n-1)$

$= n(n-1)/2$

$= \frac{n^2}{2}$

$T(n) = \Theta(n^2)$ //

* Mathematical Analysis of Recursive Algorithms

The General plan for Analysis is as shown below

* Based on Input Size, Consider the Various parameters
* Identify the Basic operation
* Obtain the no of times, Basic operation is Executed
* obtain a Recurrence Relation with Appropriate Base Condition
* Solve the Recurrence Relation & obtain the order of Growth

→ In this part of Recursive Algo Analysis, you'll learn the foll Algorithms

* Factorial of N
* Towers of Hanoi problem
* Binary representation of Decimal no
* Fibonacci Series.

* Finding the Factorial of N

* Factorial (N)

I/p → Non-Negative Integers

O/p → Value of N!

```
if n == 0
        return 1
Else
        return F(n-1) * n;
```

* Analysis :-

* The Input Size is n, we will Consider it as parameter.

* "Multiplication" is the basic operation.

* The total no of Multiplication performed can be obtained as

Let T(n) be the no of Multiplications required for function F(n)

$$\therefore F(n) = F(n-1) * n$$

$$T(n) \begin{cases} T(n) = T(n-1) + 1 & n > 0 \\ T(0) = 0 & n = 0 \end{cases}$$

$$\therefore T(n) = T(n-1) + 1 \text{ ---------- } (1)$$

$$\therefore T(n-1) = T(n-2) + 1 \text{ ------ } (2)$$

Substitute (2) in (1)

$$T(n) = T(n-2) + 1 + 1$$
$$= T(n-2) + 2$$
$$\vdots$$
$$= T(n-n) + n$$
$$= T(0) + n$$
$$= n \qquad \because T(0) = 0$$

$$T(n) = \Theta(n) \text{ //}$$

* Finding the Binary Representation of given decimal no

* Algorithm Bin(n)

I/P :- A positive decimal integer

O/P :- Binary repn

if n = 1
    return 1
Else
    Bin(⌊n/2⌋) + 1

* Analysis :-
* Time required depends only on Input Size 'n'
* Basic operation is division
* let T(n) be the no of divisions required for Input n

$\therefore T(n) = T(n/2) + 1$ & $T(1) = 0$ if $n = 1$

Consider $n = 2^k$

$$T(n) = T(2^k/2) + 1$$
$$= T(2^{k-1}) + 1$$
$$= T(2^{k-2}) + 2$$
$$\vdots$$
$$= T(2^{k-k}) + k$$
$$= T(2^0) + k$$
$$= T(1) + k$$

$T(n) = k$ ------ (1)

W.K.T $2^k = n$ Apply log on both Sides

$\log_2 2^k = \log_2 n$

$k \log_2 2 = \log_2 n$

$k = \log_2 n$ ---------- (2)

Substitute (2) in (1)

$T(n) = \log_2 n$

$\therefore T(n) = \Theta(\log_2 n)$ //

* Towers of Hanoi problem :- The problem is to transfer n discs from Source to destn.

→ The process of transfer includes the following Steps
* Transfer n-1 discs from Source to temp.
* Transfer nth disc from Source to destn

* Transfer n-1 discs from temporary to destn.

* Algorithm TowerofHanoi (n, s, t, d)

I/P:- n no of discs

O/P:- Disc Transfer to d

If (n==1)

Move disk 1 from s to d

Return

Else

TowerofHanoi (n-1, s, d, t)

Move disk n from s to d

Tower of Hanoi (n-1, t, s, d)

* Analysis :-

* The no of Moves or the time required depends only on no of discs, 'n' itself as parameter

* The Basic operation is Movement of discs

* The Recurrence Relation is calculated as shown

let T(n) be the no of Moves required for Moving n discs as per the Algo

$$T(n) = T(n-1) + 1 + T(n-1)$$

$$T(n) = 2T(n-1) + 1 \quad \& \; T(1) = 1, \; T(0) = 0$$

$\therefore T(n) = 2T(n-1) + 1$ ..... (1)

$T(n) = 2[2T(n-2)+1] + 1$ $\quad \because T(n-1) = 2T(n-2) + 1$

$= 2^2 T(n-2) + 2 + 1$

$\vdots$

$= 2^n T(n-n) + 2^{n-1} + 2^{n-2} + \cdots + 2^2 + 2 + 1$

$= 2^n T(0) + 2^{n-1} + 2^{n-2} + \cdots + 2^2 + 2 + 1$

$= 2^{n-1} + 2^{n-2} + \cdots + 2^2 + 2 + 1 \quad \because T(0) = 0$

$= \frac{1\,(2^n - 1)}{2 - 1}$ ∵ geometric progression $\sum_{n=0}^{n} a^n = \frac{a(r^n - 1)}{r - 1}$

$= 2^n - 1$

$\therefore T(n) = \Theta(2^n)$ //

* Fibonacci Series :- are the nos in the following Sequence

0, 1, 1, 2, 3, 5, 8, 13, 21 - - - - -

By defn, the first two nos are 0 & 1, then Each Subsequent nos in Series is Equal to the Sum of previous two nos.

* Algorithm Fibonacci(n)

I/P :- A non-negative Integer n

O/P :- the nth fibonacci no

```
if n ≤ 1
        return n
Else
        return F(n-1) + F(n-2)
```

* Analysis :-

* The parameter is Input Size 'n'
* The Basic operation is Addition
* let T(n) be the no of Additions required for Computing nth fibonacci no

$T(n) = T(n-1) + T(n-2)$, $T(0) = 0$ & $T(1) = 1$

$\therefore T(n) = T(n-1) + T(n-2)$ i.e $T(n) - T(n-1) - T(n-2) = 0$

Its a Characteristic Equation of form $r^2 - r - 1 = 0$

$a = 1$ $b = -1$ $c = -1$ $\therefore r_1, r_2 = \frac{-(-1) \pm \sqrt{(-1)^2 - 4.1(-1)}}{2.1}$

$= \frac{1 \pm \sqrt{5}}{2}$

$T(n) = \frac{1}{\sqrt{5}} \left\{ \left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n \right\}$

$\therefore T(n) = \Theta(\phi^n)$ //

$\therefore \frac{1+\sqrt{5}}{2} = 1.6180 =$ Golden ratio (ϕ)

* Important problem types :-

There are some common problems that we come across while computing. Usually many of the problems can be categorized into any one of these major problem Types

The Most Important problem types are

* Sorting → The problem involving the rearrangement of the items of a given list in some particular order is known as "Sorting problem".

→ The Sorting makes some task Easier such as Searching & comparing two Elements Etc.

→ There are plenty of Algo Available for Sorting, none of these best suits for all possible Situations.

Ex:- Bubble Sort, Selection Sort, Mergesort Etc.

* Searching → The Searching problem deals with Searching of a given value called "Key" in the given list.

→ There are Several Algorithms Available for Searching, no one suites best for all the cases.

→ Few of them work Efficiently better on Sorted list & some Algo requires Additional Memory

Ex: Linear Search, Binary Search Etc.

* String processing → In rapid growing Applications of Computers Science, processing of text takes an Important role.

Searching for a particular word in a text is known as "String processing".

Ex: KMP String Matching Algo, Naïve String Matching Algo Etc.

* Graph problems :- Graph is a set of points known as "vertices" some of which may be connected by a line/curve called "Edges"

→ Graphs are helpful in problems like transportation & communication Networks, project Management Etc.

→ The Basic graph Algorithms include traversal Algo, Shortest path Algo Etc

Ex:- Bellman-ford Algo, Dijakstras Algo Etc

* Combinatorial problems :- The problems involving Combinatorial Elements like permutation, Combinations, sets Etc are known as "Combinatorial problems".

→ The difficulty with these problems is the Combinatorial objects grows Extremly fast with problem Size

→ There is no Such Algo for giving Accurate Soln within Feasible Amount of time.

* Geometric problems :- We will come Across geometric problems in the field of Computers graphics & Robotics Etc

→ Very- well known problem is "closest-pair" problem that is used to find the closest pair of points Among the given n points.

Ex:- Line-clipping Algo, polygon-clipping Algo Etc

* Numerical problems :- Involves Solving System of Equations, Comp-uting definite Integrals, Etc

→ Many of the numerical problems can be Solved Approximately as these problems involves real values.
Using the round-off technique, Many Variables will loose their original value.

Ex:- Euclide Algo, primality testing Algos Etc