11/2/16

MODULE - 1

Algorithm :- It is an unambigious, sequence of steps to solve the problem,
ie it generates the required o/p by accepting legitimate i/p in finite amount of time.

Characteristics of Algorithm :-

1. Non-ambigious steps
2. Range of inputs [It should accept]
3. Legitimate [valid] input.
4. Finiteness.
5. Definiteness [correctness, it definitely gives the o/p]
6. Effectiveness
   Some Algorithm can be represented in different ways
   i) Natural language rep^{-n} [English]
   ii) pseudo code rep^{-n}
   iii) Flow chart method
7. Same problem can be solved using many idea
   ie Design techniques,

GCD :-
1) Euclids method
2) Consecutive integer check method.
3) middle school method.

Effectiveness :- Every step should be effective [Important] to generate the required o/p

1. Euclids - GCD

Algorithm :- Euclids ± GCD (m,n)

// Input : non-negative integer m,n and both should not be zero

// output : GCD of m,n.

while ($n \neq 0$)

$r \leftarrow m \% n$

$m \leftarrow n$

$n \leftarrow r$.

endwhile

return (m)

2. Consecutive integer check method ~~language~~

Algorithm :- consecutive integer check method ~~language~~ - GCD

m=30, 4

Positive (non zero)

//Input :- ~~non-negative~~ integers m,n and both should ~~be~~ not be zero

t	m%t	n%t
4	2	-
3	0	1
2	0	0
GCD		

//ouput :- ~~output~~ GCD of m,n.

step 1 : $t \leftarrow \min(m,n)$

step 2 : Divide m by t. if remainder is non-zero, then go to step 4

step 3 : Divide n by t. if remainde is zero, return t as GCD

step 4 : Decrease t by 1 ; Goto step 2.

Pseudo-code

1. select minimum value of two inputs assign it to t.

2. divide ~~m~~ m and n by t, if both remainder zero, then t is GCD.

3. else decrement t by 1 and repeat step 2 until both remainder are zero.

5. Middle school method

Algorithm : middle school - GCD (m,n)
// input : non-zero positive integer
// output : GCD of m,n.
step 1 : Find the factors of m
step 2 : Find the factors of n
step 3 : Find the common factors from step1 and step2
step 4 : return largest common factor from step3 as GCD

```
#include <iostream.h>
```

Sieve of Eratosthemes

used to generate prime numbers for given range.

n=2

Initial -
2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20

Sieve (2) 2, 3, 5, 7, 9, 11, 13, 15, 17, 19

Sieve (3) 2, 3, 5, 7, 11, 13, 17, 19.

Algorithm sieve (n)

// input : An integer $n \geq 2$.
// output : list A contain prime number upto n.

```
for p ← 2 to n do
    L[p] ← p
end for
for p ← 2 to ⌊sqrt(n)⌋ do
    j ← p * p ;
```

```
        if(L[j] ≠ 0)
            while(j ≤ n) do
                L[j] ← 0
                j ← j + p
            endend while
        end if
    end for
        j ← 1
    for p ← 2 to n do
        if(L[p] ≠ 0)
            A[j] ← L[p]
            j ← j+1
        end if
    end for
    return (A)
```

Algorithm specification :-

1) comment is specified through //
2) Algorithm header is specified as
   Algorithm Name (parameter list)
3) ~~Alg~~ specify algorithms legitimate inputs and the required O/P
4) compound statements are specified with { }
   or begin ..... end
5) Record data type are specified through { }
   {Like Structure}

   Ex: record student
   { {
       name datatype

   }

(Compound data types) each of them are accessed their Instances.

6) if (condition) then
    statement 1
    statement 2
    :
  ~~else~~.
else.
    statement
    :
endif.

7. loop statement

① while (condition) do
    :
  endwhile.

② repeat } Do while
    :
  until (condition

③ for variable ← value1 to value2 do
    :
  end for

④ for variable ← value1 docon to value2 do

⑤ for variable ← value1 to value increase by value do.

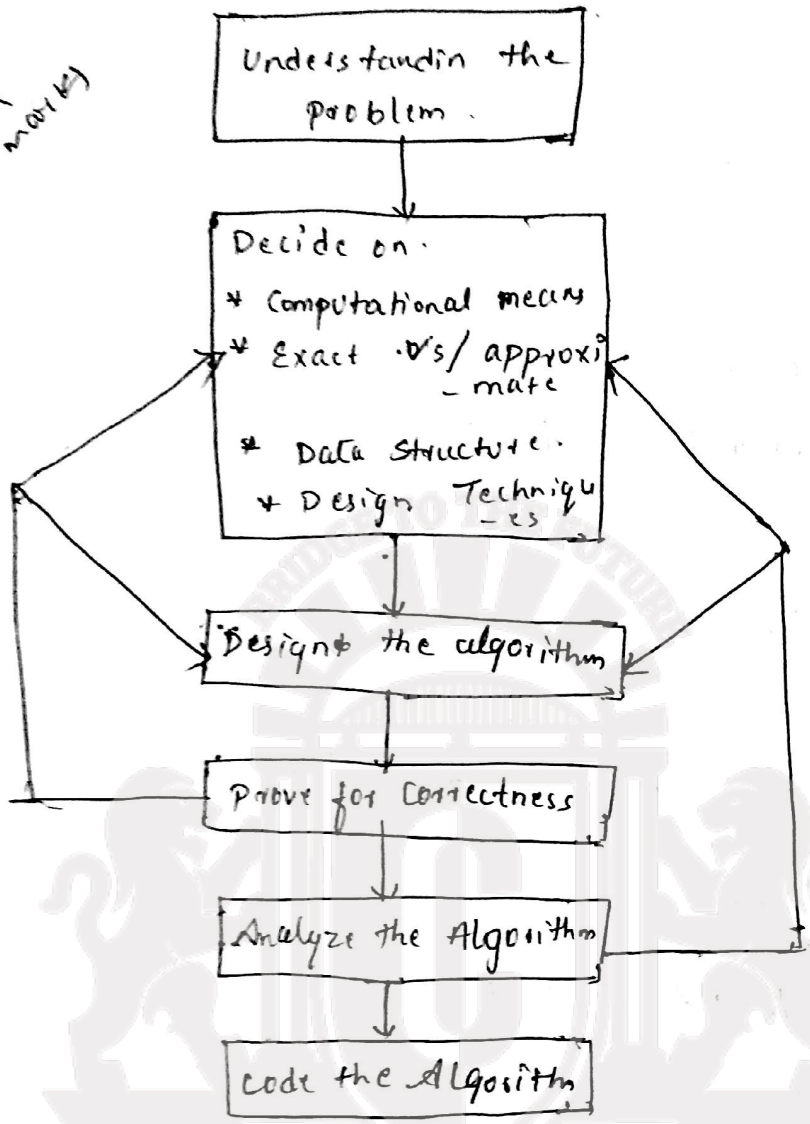
6) Assignment operator (←)
   eg:- A ← B.

⑦ Relational operator (<, ≤, >, ≥, =, ≠).

⑧ Multidimentional array index are specified in [and].
   ~~in [and]~~.
   Ex:- A[i, j]

Algorithm design and analysis process / Frame work

Q- 10 marks



1. The first step involves in the throughly understanding the problem statement and trying to solve the problem manually for the different instances (copies) of the problem.

2- Decide on

→ Computational means

The computational capability judges the computing powers of the device on which the algorithm is made to run, common factors considered were speed and memory. The type of the algorithm can be considered as sequential algorithm and parallel algorithm

→ Exact v/s Approximate solving : It decides on whether the problem should generate exact sol^n or approximate

sol^{-n}. Generating a exact sol^{-n} may be difficult for the following instances.

a) generating o/p for non-linear eq^{-n}s, solving integrals, calculating square roots.

b) Due to complexity of solving problem exactly which requires high computation capability outcomes are compromised with approx values.

→ Data Structure

The choice of data structure affects the performance of the algorithm. Choose the data structure that are comprised with ~~approx values~~ approximate for current problem

ex:- Choosing array instead of linked list.

⇒ Algorithm Design techniques

Designing an Algorithm is defined as the method or the strategy to solve the given problem and get the desired o/p.. A few design techniques are listed below.

i) Brute force - Linear search, bubble sort,...
ii) Divide & conquer
iii) Decrease and conquer.
iv) Greedy technique.
v) Dynamic programming.
vi) Back tracking, Branch and Bound.

3) Design of Algorithm

Designing algorithm involves representing each and every step with non-ambiguity features & simple and basis. Algorithm can be represented using natural language, flowchart or pseudocode.

4) Prove the correctness / Algorithm valuation

It is the process of checking a correctness of the algorithm for the range of legitimate input and their respective output. Approximate sol^{-n} may be verified by checking their error way, which should not exceed to a certain limit.

Analysis Framework :-

Based on the resource of computing machines.

1) CPU time → Time complensity Analysis.
2) RAM space → space complensity Analysis.

---

1) measuring input size

Identify potential input for the algorithm which effects the time efficiency.

ex:- linear search (A[1 ....n], key)

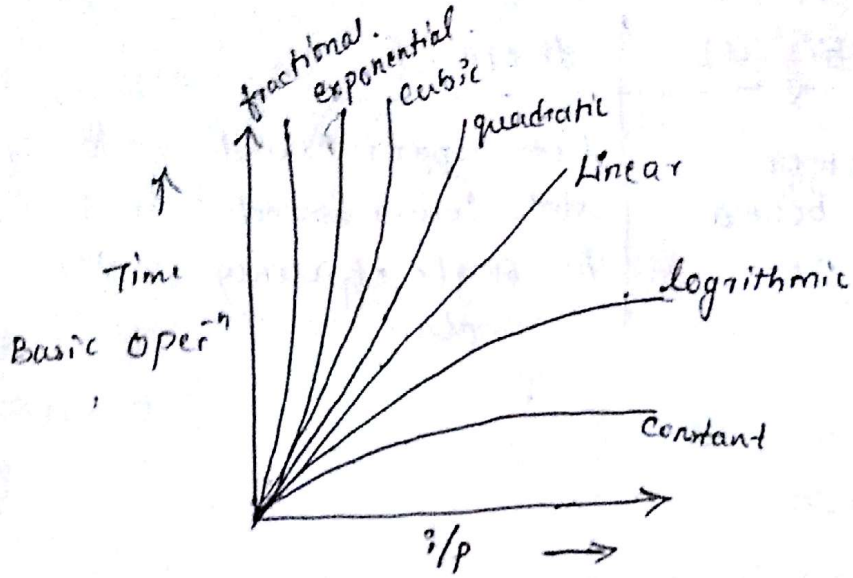
In this size n is a prominant i/p parameter which effects the efficiency of the algorithm.

2) Unit of measuring time
   i) clock rate - device time [we dont conside] ✗
   ii) primitive operation - counting [— " —] ✗
   iii) Basic operation ✓

5) Analyses of the algorithm :-
The algorithm is analysed based on time and space constrained and an algorithm which suits the best for the application choosen.

6) Code the Algorithm :-
Algorithm is implemented on machine using one of the programming languages.

3. Order of growth
It is the relationship blw i/p size and the time consumed for running an algorithm as the input size increases, the time consumed proportionately increases in a particular order.
The order in which time increases w.r.t input is called as order of the growth.



4. efficiency classes [standard orderd of growth]

Name	Representation	Input n=10
constant	c	c
logrithmic (binary search)	$\log_2 n$	$\log_2 10 = 3.32$
Linear (addition)	n	n = 10
n-log-n	nlogn	$10\log_2 10 = 33.2$
Quadratic [Bubble sort]	n^2	100
cubic [matrix mul]	n^3	1000
exponential	2^n	1024
Factorial	n!	3628800

Time complexity ↓

5. Asymptotic notations

It is a symbolic representation to categorise the algorithmic effeciency at different instances of input of the same size, comparing in the standard efficiency classes

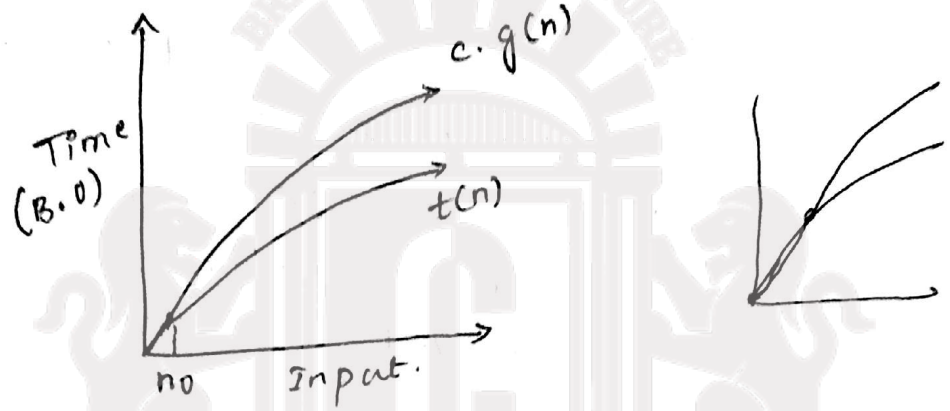
Big-O

A fn $t(n)$ is said to be in $O(g(n))$, denoted by $t(n) \in O(g(n))$, if $t(n)$ is bounded above by some constant multiple of $g(n)$ $\forall$ large 'n', ie if there exist some positive constant 'c' and some non negative integer 'n_0' such that $t(n) \leq c\,g(n)$ $\forall\, n \geq n_0$

$$\boxed{t(n) \leq c \cdot g(n) \quad \forall\, n \geq n_0}$$

$t(n) \rightarrow$ Order of growth of algorithm.

$g(n) \rightarrow$ Refrence order of growth.



Ex:-

P.T $100n + 5 \in O(n)$

$t(n) = 100n + 5$

$g(n) = n$

Proof:-

If $100n + 5 \in O(n)$, then

$100n + 5 \leq c \cdot n \quad \forall\, n \geq n_0$.

$100n + 5 \leq 100n + n$

$100n + 5 \leq 101n$.

$$\boxed{c = 101}$$

$100n + 5 \leq 101n$.

$n = 1 \quad 100 + 5 \leq 101 \quad \times$

$n = 2 \quad 100 \times 2 + 5 \leq 101 \times 2 \quad \times$

$$\boxed{n = 5 \quad 100 \times 5 + 5 \leq 101 \cdot 5} \checkmark$$

$n = 6 \quad 100 \times 6 + 5 \leq 101 \times 6$ ✓

$\boxed{n_0 = 5}$

$\therefore 100n + 5 \leq 101n \quad \forall n \geq 5.$

$\therefore \boxed{100n + 5 \in O(n)}$

Ex 2 : $3n^2 + n + 2 \in O(n^2)$

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

$g(n) = n^2$

Proof : If $3n^2 + n + 2 \in O(n^2)$, then

$3n^2 + n + 2 \in c \cdot n^2 \quad \forall n \geq n_0$

$3n^2 + n + 2 \in 3n^2 + 1$

$3n^2 + 2n \in 3n^2 + n + n^2$

$3n^2 + 2n \in 4n^2 + n$

$\boxed{c = 4}$

$n = 1$	$3n^2 + 2n$	$\in 4n^2 + n$
	6	5
$n = 2$	16	18

$\boxed{n_0 = 2}$

$3n^2 + n + 2 \leq 4n^2 + n \quad \forall n \geq 2.$

$\boxed{3n^2 + n + 2 \in O(n^2)}$

Ex 3 : $4n^3 + 3 \in O(n^3)$

$f(n) = 4n^3 + 3$

$g(n) = n^3$

Proof: If $4n^3+3 \in O(n^3)$ then.

$4n^3+3 \in 4n^3+n^3$.

$5n^3$

$c = 5$ ✓

$4n^3+3 \in 4n^3+n^3$

$4n^3+3 \in 5n^3$

$c = 5$

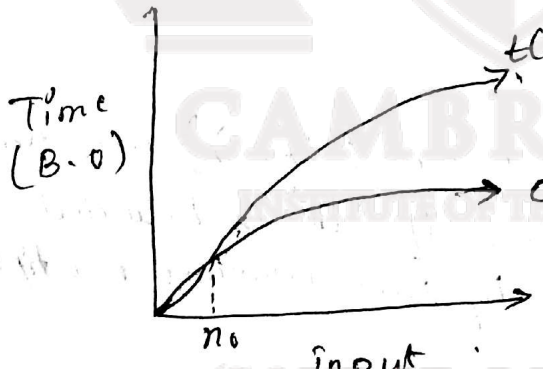
$n_0 = 2$

Big - Ω [omega]

a function $t(n)$ is said to be in $\Omega(g(n))$ denoted by $t(n) \in \Omega(g(n))$, if $t(n)$ is bounded below by some constant multiple of $g(n)$ $\forall$ large n.

ie: if there exists some +ve constant 'c' and some non-negative integer n_0 such that

$$t(n) \geq c \cdot g(n) \quad \forall \; n \geq n_0$$



ex:-

P.t $3n^2+n \in \Omega(n)$

$t(n) = 3n^2+n$.

$g(n) = n$.

$3n^2+n \geq c.n \quad \forall \; n \geq n_0$

$3n^2+n \geq 3.n+n$.

$3n^2+n \geq 4n$.

$\boxed{c=4}$

if $n_0 = 1$. $3n^2+n \geq 4n \quad \forall n \geq 1$

hence $\boxed{3n^2+n \in \Omega(n)}$.

Ex:. P.T $4n^3+2n^2+n+5 \in \Omega(n^3)$

$t(n) = 4n^3+2n^2+n+5$

$g(n) = n^3$

$4n^3+2n^2+n+5 \geq c.n^3$

$4n^3+2n^2+n+5 \geq 4n^3+2n^2+n^3$

$4n^2+2n^2+n+5 \geq 7n^3$

$c=7$

$n=1$, $4+2+1+5 \geq 7$

$12 \geq 7$

$n_0 = 1$

$4n^3+2n^2+n+5 \geq 7 \quad \forall n \geq 1$

$4n^3+2n^2+n+5 \geq \Omega(n^3)$.

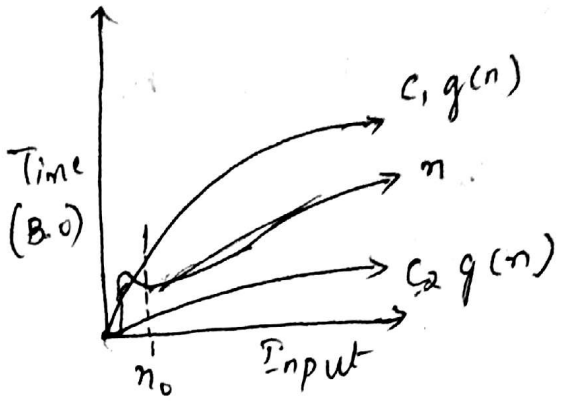
21/2/17

Theta - Θ

A fn $t(n)$ is said to be $\Theta(g(n))$ denoted by $t(n) \in \Theta(g(n))$, if $t(n)$ is bounded both above and below by some constant multiples of $g(n)$ $\forall$ large n.

~~$\forall$ c~~ i.e if there exist some +ve constants c_1 and c_2 and some non negative integer n_0 such that

$c_2 g(n) \leq t(n) \leq c_1 g(n) \quad \forall n \geq n_0$



P.T. $\frac{1}{2} n(n-1) \in \Theta(n^2)$

$t(n) = \frac{1}{2} \cdot n(n-1)$.

$g(n) = n^2$

$c_2 \cdot n^2 \leq \boxed{\frac{1}{2} n(n-1)} \leq c_1 n^2 \quad \forall n \geq n_0$

(1) ← $\frac{1}{2} n(n-1)$ → (2)

eq^n (1).

$c_2 n^2 \leq \frac{1}{2} n(n-1)$

$\frac{1}{2} n(n-1) \geq c_2 \cdot n^2$

$\frac{1}{2} n^2 - \frac{1}{2} n \geq c_2 n^2$

$\frac{1}{2} n^2 - \frac{1}{2} n \geq \frac{1}{2} n^2 - \frac{1}{2} \cdot \frac{1}{2} n^2$.

$\geq \frac{1}{2} n^2 - \frac{1}{4} n^2$

$\frac{1}{2} n(n-1) \geq \frac{1}{4} n^2$

$c_2 = \frac{1}{4}$

$n = 1 \quad \frac{1}{2} \cdot 1(1-1) \geq \frac{1}{4} \cdot 1^2$ ✗

$n = 2 \quad \frac{1}{2} \cdot 2(2-1) \geq \frac{1}{4} 2^2$ ✓

$n = 3 \quad \frac{1}{2} \cdot 3(3-1) \geq \frac{1}{4} 3^2$ ✓

$\boxed{n_{01} = 2}$

Consider eqn (2)

$\frac{1}{2} n(n-1) \leq c_1 n^2 \quad \forall \quad n \geq n_{02}$

$\frac{1}{2} n^2 - \frac{1}{2} n \leq c_1 \cdot n^2$

$\frac{1}{2} n^2 - \frac{1}{2} n \leq \frac{1}{2} n^2$

$c_1 = \frac{1}{2}$

$n = 1 \quad \therefore \quad \frac{1}{2} - \frac{1}{2} \leq \frac{1}{2}$

$n = 2 \quad \therefore \quad 2 - 1 \leq 2$

$\boxed{n_{02} = 1}$

$n_0 = \max\{n_0, n_{02}\}$

$\max\{2, 1\}$

$= 2.$

$\therefore \frac{1}{4} n^2 \leq \frac{1}{2} n(n-1) \leq \frac{1}{2} n^2 \quad \forall \quad n \geq 2.$

$\therefore \boxed{\frac{1}{2} n(n-1) \in \theta(n^2)}$

2. $100n + 5 \in \theta(n).$

~~$c_2 \cdot n \leq \frac{1}{2} n(n$~~

$\boxed{c_2 n \leq 100n + 5} \leq c_1 n$

$c_2 n \leq 100n + 5$

$100n + 5 \geq c_2 n.$

$100n + 5 \geq 100n$

$100n + 5 \geq 100n.$

$c_2 = 100$

$n = 1$

$n = 2$

$n = 3$

$n = 4$

$n_0 =$

$\therefore 100n + 5 \leq c_1 n$.

$c_1 n \geq 100n + 5$

~~100~~ $100n + 5 \leq c_1 n$.

$100n + 5 \leq 100n + n$

$\boxed{c_1 = 101}$.

$n = 5$, ✓

$\boxed{n_{o_2} = 1}$.

$n_0 = \max\{n_{01}, n_{02}\}$

$\max\{5, 1\}$

$\{5\}$

$\therefore 100n \leq 100n + 5 \leq 101n$.

$\boxed{100n + 5 \in \Theta n}$

Properties of Asymptotic notations

1. If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$ then $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$

Proof :-

If $t_1(n) \in O(g_1(n))$ then $t_1(n) \leq c_1 g_1(n)$ $\forall n \geq n_{01}$

Ily

$= t_2(n) \in O(g_2(n))$ then, $t_2(n) \leq c_2 \cdot g_2(n)$ $\forall n \geq n_{02}$.

$= t_1(n) + t_2(n) \leq c_1 \cdot g(n) + c_2 g_2(n)$

To maximize RHS

$c_3 \cdot \max\{c_1, c_2\}$

Replace c_1 and c_2 by c_3

$= t_1(n) + t_2(n) \leq c_3 \cdot g_1(n) + c_3 \cdot g_2(n)$

Choose $\max\{g_1(n), g_2(n)\}$ to replace $g_1(n)$ and $g_2(n)$.

$= t_1(n) + t_2(n) \leq c_3 \cdot \max\{g_1(n), g_2(n)\} + c_3 \cdot \max\{g_1(n), g_2(n)\}$

$\leq 2 c_3 \cdot \max\{g_1(n), g_2(n)\}$

$c = 2c_3$

$= t_1(n) + t_2(n) \leq c \cdot \max\{g_1(n), g_2(n)\}$.

$n_0 = \max\{n_{01}, n_{02}\}$.

$= \therefore t_1(n) + t_2(n) \leq \max\{g_1(n), g_2(n)\} \; \forall n \geq n_0$

hence.

$$\boxed{t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})}$$

2. comparing 2 different order of growth $(t(n), g(n))$ using limits

ie $\lim_{n \to \infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & t(n) \text{ is smaller order of growth than } g(n) \\ c, \text{ where } c > 0 & t(n) \text{ is same order of growth as } g(n) \\ \infty & \end{cases}$

$\infty \rightarrow t(n)$ has higher order growth than $g(n)$

4/2/17

Ex:- Compare $n!$ with 2^n using limits

$$\lim_{n\to\infty} \frac{n!}{2^n}$$

using sterling formula for $n!$

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

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

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

$$= \infty$$

∴ $n!$ has higher order of growth compared to 2^n (exponent)

Mathematical Analysis on non-Recursive Algorithm

1) Decide on input parameter
2) Identify the basic operation
3) Check whether the algorithm depends only on input or if these are any variation, if so estimate Best case, worst case and average case time efficiency separately.
4) Build summation equation for no of Basic operation executed.
5) solve the equation & ascertain it to one of the standard efficiency class

Ex:-

```
Algorithm liner_search (A[1....n], key).
//Input : An array of integers A[] with n elements and key
//output : returns true if found else false.

    for i ← 1 to n do
        if (A[i] == key) then
            return True
        end if
    end for
    return false.
```

Analysis

1. Input parameter - 'n' size of array A.
2. Basic operation - Search / comparison - A[i] = key
3. Apart from input size 'n' the algorithm produces varying order of growth, therefore estimate time analysis for best case, worst case and average cas seperately

4) Best case : if the key is found at first position. comparision is one

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

s i] $C_{Best}(n) \in \Omega(1)$ => constant order of growth

Worst case : if the key is found in last position.

$$C_{worst}(n) = \sum_{i=1}^{n} 1$$

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

$$= n //$$

$\sum_{i=l}^{u}$ constant

$= \text{constant} \times (u-l+1)$

sii] $C_{worst}(n) \in O(n)$ => linear order of growth

Average case :

$$C_{Avg}(n) = \frac{(1+2+3+4+\ldots+n)}{n} * p + (1-p) * n.$$

$$= \frac{1}{n}\left(\frac{(n+1)*n}{2}\right) * p + (1-p) * n$$

$$= \left(\frac{n+1}{2}\right) * p + (1-p) * n$$

If key is found, $p = 1$.

$$C_{Avg-found}(n) = \left(\frac{n+1}{2}\right) * 1 + (1-1) * n.$$

$$= \frac{n+1}{2} \approx n/2 + 1/2.$$

For last value of n.

$$C_{Avg-found}(n) \approx 1/2\, n.$$

$C_{Avg-found}(n) \in \Theta(n)$

if key is not found, $p = 0$.

$C_{avg-fail}(n) = \left(\frac{n+1}{2}\right) * 0 + (1-0) * n$

$= n$.

$\boxed{C_{avg-fail}(n) \in \theta(n)}$ => constant, order of growth.

Ex 2 Finding the largest timing in given list.

Algorithm max-element (A[1....n])
//Input: An array A[] of n integers (positive)
//output: return MAX.

```
MAX ← 0.
for i ← 0 to n do
    if [A[i] > MAX]
        MAX ← A[i]
    end if
end for
return MAX
```

Analysis

1) Input parameter → 'n' size of array 'A'
2) Basic operation → search / comparision - A[i] > MAX
3) the algorithm completely depends on 'n'
   Hence no variations.
4) $C(n) = \sum_{1}^{n} 1$

$= n$ linear order of growth.

$C(n) \in \theta(n)$.

4. $$C(n) = \sum_{i=1}^{n} 1$$

$$= n - 1 + 1$$

$$= n$$

$$\boxed{C(n) \in \Theta(n)}$$

Exp:- To find the uniqueness of a given list

Algorithm Uniqueness (A[1...n])

// Input : An Array A[] of 'n' integer

// output : Unique list / duplicate list .

```
for i ← 1 to n-1 do.
    for j ← i+1 to n do.
        If [A[i] = A[j]) then
            return (duplicate).
        end if
    end for
return (unique).
```

Analysis :- (check uniqueness).

(1) Input parmeter :- 'n' Size of array

(2) Basic operation :- comparasion - A[j] = A[j]

(3) for Uniqueness check algorithm depends only on 'n' :- no variation.

(4) $$C(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1$$

(5) $$C(n) = \sum_{i=1}^{n-1} (n-(i+1)+1) \Rightarrow \sum_{i=1}^{n-1} (n-i-1+1)$$

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

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

$$C(n) = n(n-1-1+1) - \frac{(n-1)(n-1+1)}{2}$$

$$C(n) = n(n-1) - \frac{n(n-1)}{2}$$

$$C(n) = \frac{n(n-1)}{2}$$

$$C(n) = \frac{n^2}{2} - \frac{n}{2}$$

for large value of n.

$$\boxed{C(n) \in \Theta(n^2)}$$

eg Matrix multiplication

$$A = \begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} \quad B = \begin{bmatrix} 3 & 2 \\ 1 & 7 \end{bmatrix}$$

Algorithm :-

```
matrix - multiplication (A[n,n], B[n,n]).
// Input :- matrix A[n,n], B[n,n] of integers.
// output :- Resultant matrix C[n,n].
// Assumed that matrix C is initialized to zero
    for i ← 1 to n do
    for j ← 1 to n do.
    for k ← 1 to n do
     C[i,j] ← C[i,j] + A[i,k] * B[k,j].
      end for
      end for
      end for
    return C.
```

Analysis :-

1) Input Parameter - 'n' nxn order of matrix (Both A or B).

2) Basic operation :- multiplication
- A[i,k] * B[k,j].

3) Depends only on order of matrix - 'n' no variations.

4) $M(n) = \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} 1$.

~~M(n) = (~~

$M(n) \in (n^3)$

cubic order of growth.

Ex: counting no of Bits required to represent Decimal no

Algorithm :- Bits(n)

Input : Decimal no 'n' → non-negative integer

Output : count ↔ no of bits to represent 'n'

```
count ← 1
while (n>1) do
  n ← n/2
  count ← count++
end while
return (count)
```

Analysis :-

1) Input parameter :- 'n'

2) Basic operation :- Addition

3) Depends only on 'n' - no variations

4) $A(n) = \sum_{i=1}^{\lfloor \log_2 n \rfloor} 1$

$\lfloor \log_2 n \rfloor$ = upper (m − lower (m + 1

$\lfloor \log_2 n \rfloor$. $\lfloor \log_2 n \rfloor - 1 + 1$ $\lfloor \log_2 n \rfloor$

$\lfloor \log_2 n \rfloor$ $\boxed{A(n) \in \Theta(\log n)}$

Logrithmic order of growth.

:. Mathametical analysis for recursive algorithm :-

1) Decide on input parameter
2) Identify Basic operation
3) Check if the order of growth depends only on 'n' or if there are any variation, if so, then estimate Best case, worst case and Average case seperately
4) Build Recurrance relation based on the Basic operation.
5) Solve the recurrence relation and ascertain it to one of the standard efficiency class

Ex :-

```
Algorithm Factorial (n)
// Input : non negative integer n
// Output : factorial of n
        if (n≤1)
            return (1)
        else
            return (n * factorial (n-1))
```

Analysis :-

1) Input parameter is 'n'
2) B.O → multiplication - n * factorial(n-1)

3. Depends only on 'n' - no variation

4.

$$\text{factorial}(n) = \begin{cases} 1 & n \le 1 \\ n * \text{factorial}(n-1) & n > 1 \end{cases}$$

basic operation.

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

$M(n) = 1 + M(n-1)$ until $M(1) = 0$.

5) Solve using Backward substitution method

{ Backward substitution method }

$m(n) = 1 + m(n-1)$.

$= 1 + 1 + m(n-2)$.

$= 2 + m(n-2)$

$= 2 + 1 + m(n-3)$.

$= 3 + m(n-3)$

$n-1 + m(n-(n-1))$. $\{ m(1) = 0$

$= n - 1$

For value of n, $m(n) \approx n$.

or

$$\boxed{m(n) \in \Theta(n)}$$

linear order of growth.

Finding no of bits to represent Decimal number

Algorithm :- Bit-count (n).

//Input :- non-negative integer 'n'

// output :- count of no of bits required.

```
if (n ≤ 1) then.
    return (1)
else
    return (1 + Bit_count (⌊n/2⌋)).
end if
```

Analysis :-

1) Input parameter - 'n'

2) Basic operation - addition "1+ Bit_count($n/2$)

3) Depends only on 'n' - no variations

4) Algorithms recurrence relation.

$$Bit\_count(n) = \begin{cases} 1, & \text{if } n \leq 1 \\ 1 + Bit\_count(n/2) & n > 1 \end{cases}$$

B.O recurrence relation.

$$A(n) = \begin{cases} 0 & n \leq 1 \\ 1 + A(n/2) & n > 1 \end{cases}$$

5) $A(n) = 1 + A(n/2)$ with $A(1) = 0$.

$1 + [1 + A(n/4)]$

$1 + [1 + [1 + A(n/8)]]$

:

complicated.

Consider $n = 2^k$ ⟶ (1)

$\therefore A(n) = 1 + A(2^k/2)$

$= 1 + A(2^{k-1})$

$= 1 + [1 + A(2^{k-2})]$

$= 2 + A(2^{k-2})$

$3 + A(2^{k-3})$

⋮

$k + A(2^{k-k})$

$A(n) = k$.

Apply $\log_2$ on eq^n (1)

$\log_2 n = k \log_2 2$

$\log_2 n = k$.

$\log_2 n = A(n)$.

$\boxed{A(n) \in \Theta(\log n)}$

logrithmic order of growth.

~~Tim~~

Tower of Hanoi

Algorithm :- Tower (n, S, T, D)

// Input : no of discs in' input 3 poles - S, T, D

//output : n-discs on destination pole

```
if (n = 1) then
    move disc 1 from S to D
else
    Tower (n-1, S, D, T)
    move nth disk from S to D
```

Tower (n-1, T, S, D)

end if.

Analysis

1) input parmeter - 'n' no of disks.

2) B.O -> moving discs.

3) Depends only on 'n' - no variation.

4) Algorithms .. recurrence relation.

$$\text{Tower}(n,S,T,D) = \begin{cases} \text{move disc 1 from s to D} & n=1 \\ \text{Tower}(n-1,S,D,T), \\ \text{move } n^{th} \text{ disc from s to D} \\ \text{Tower}(n-1,T,S,D) & n>1 \end{cases}$$

B.O. recurrence relation

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

5) $m(n) = 1 + 2M(n-1)$ until $M(1)=1$

$= 1 + 2[1 + 2M(n-2)]$

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

$= 1 + 2 + 2^2 [1 + 2M(n-3)]$

$= 2^0 + 2^1 + 2^3 M(n-3)$

$= 2^0 + 2^1 + 2^3 + 2^4 M(n-4)$

$= 2^0 + 2^1 + 2^3 + \cdots + 2^{n-1} M(n-(n-1))$

$= 2^0 + 2^1 + 2^3 + 2^4 + \cdots + 2^{n-1}$

= GP

= sum of Geometric progression.

$$S_n = \frac{a.(r^n - 1)}{r-1}$$

$a = 1, \; r = 2$.

$$\therefore M(n) = \frac{1(2^n - 1)}{2-1} \Rightarrow 2^n - 1$$

For large value of n.

$M(n) \approx 2^n$.

$\therefore \boxed{M(n) \in \Theta(2^n)}$

4/3/17 Exponential order of growth.

Fibonacci series

Seed elements - 0, 1

Algorithm Fib-iterative (n)
//input : an integer $n \geq 2$.
//output : n^{th} term of Fib series
// Auxillary array F[0...n]

```
F[0] ← 0
F[1] ← 1
for i ← 2 to n do
    F[i] ← F[i-1] + F[i-2]
end For.
return;
```

Analysis

1) Input parameter - n.
2) B.O ⇒ Addition - F[i-1] + F[i-2].
3) Depends only on 'n' no variations
4) $A(n) = \sum_{i=2}^{n} 1$

$A(n) = n-2+1$

$= n-1$

For large value of n.

$A(n) \approx n$.

$\boxed{A(n) \in \Theta(n)}$

Linear order of growth

Recursive method

$f(n) = f(n-1) + f(n-2)$

$f(n) - f(n-1) - f(n-2) = 0 \quad \longrightarrow (1)$

Similar to $a\,x(n) + b\,x(n-1) + c\,x(n-2) = 0$.
homogenous linear second order equation.

Its characteristic eqn

$$ar^2 + br + c = 0.$$

$$r = \frac{-b \pm \sqrt{b^2-4ac}}{2a}.$$

Apply to eqn (1).

$a = 1, \; b = -1, \; c = -1$

Characteristic eqn of $f(n)$

$$= r^2 - r - 1 = 0.$$

$$= \frac{-(-1) \pm \sqrt{(-1)^2 - 4(1)(-1)}}{2(1)}$$

$$= \frac{+1 \pm \sqrt{1+4}}{2}$$

$$r = \frac{+1 \pm \sqrt{5}}{2}.$$

$$r_1 = \frac{1+\sqrt{5}}{2} \qquad r_2 = \frac{1-\sqrt{5}}{2}.$$

$$x(n) = \alpha r_1^n + \beta r_2^n.$$

$$f(n) = \alpha\left(\frac{1+\sqrt{5}}{2}\right)^n + \beta\left[\frac{1-\sqrt{5}}{2}\right] \longrightarrow (2)$$

We know that

$f(0) = 0, \quad f(1) = 1$

$$f(0) = \alpha\left(\frac{1+\sqrt{5}}{2}\right)^0 + \beta\left[\frac{1-\sqrt{5}}{2}\right]^0 = 0.$$

$$\alpha + \beta = 0 \qquad \boxed{\beta = -\alpha}.$$

Consider $f(1) = 1$ in eq^{n} (2).

$$f(1) = \alpha\left[\frac{1+\sqrt{5}}{2}\right]^1 + \beta\left[\frac{1-\sqrt{5}}{2}\right]^1 = 1.$$

Replace $\beta = -\alpha$.

$$f(1) = \alpha\left[\frac{1+\sqrt{5}}{2}\right] - \alpha\left[\frac{1-\sqrt{5}}{2}\right] = 1$$

$$\alpha\left[\frac{1+\sqrt{5}-1+\sqrt{5}}{2}\right] = 1$$

$$\alpha\left[\frac{2\sqrt{5}}{2}\right] = 1$$

$$= \alpha[\sqrt{5}] = 1$$

$$\boxed{\alpha = \frac{1}{\sqrt{5}}}$$

$$\boxed{\beta = -\frac{1}{\sqrt{5}}}$$

$$f(n) = \frac{1}{\sqrt{5}}\left[\frac{1+\sqrt{5}}{2}\right]^n - 1/\sqrt{5}\left[\frac{1-\sqrt{5}}{2}\right]^n.$$

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

Golden Ratio $\phi = \frac{1+\sqrt{5}}{2}$.

$\hat{\phi} = \frac{1-\sqrt{5}}{2}$.

$$f(n) = 1/\sqrt{5}\left[\phi^n - \hat{\phi}^n\right]$$

~~Analysis~~

Algorithm :- Fib - rec (n)

// Input : An integer $n \geq 2$.

if ($n \leq 1$)

return (n)

else
    return (Fib-rec(n-1) + Fib-rec(n-2)).
endif

Analysis

1) Input parameter - 'n'
2) B.O. Addition Fib-rec(n-1) + Fib-rec(n-2).
3) Depending only on 'n' - no recursions.
4) Algorithm recurrence relation.

$$Fib\text{-}rec(n) = \begin{cases} 0, 1 & n \le 1 \\ Fib\text{-}rec(n-1) + Fib\text{-}rec(n-2) & n > 1 \end{cases}$$

Based on B.O.

$$A(n) \begin{cases} 0 & n \le 1 \\ A(n-1) + A(n-2) + 1 & n > 1 \end{cases}$$

5) $A(n) = A(n-1) + A(n-2) + 1$

$A(n) - A(n-1) - A(n-2) - 1 = 0.$

$(A(n)+1) - (A(n-1)+1) - (A(n-2)+1) = 0.$

$B(n) = A(n)+1$, $B(n-1) = A(n-1)+1$

$B(n-2) = A(n-2+1).$

$\therefore B(n) - B(n-1) - B(n-2) = 0.$

Same as $f(n) - f(n-1) - f(n-2) = 0.$

Sol^{n} for $f(n) = 1/\sqrt{5}(\phi^n - \hat{\phi}^n)$

$\therefore$ Applying the same solution

$$B(n) = 1/\sqrt{5}(\phi^n - \hat{\phi}^n).$$

$B(n) = A(n) + 1$

$A(n) = B(n) - 1$

$$\therefore A(n) = \frac{1}{\sqrt{5}}(\phi^n - \hat{\phi}^n) - 1$$

For large values of n, subtracting 1 becomes negligible and $\hat{\phi}^n$ is inverse of ϕ^n, so it can be negligible.

$$A(n) \approx \frac{1}{\sqrt{5}} \phi^n.$$

$$\boxed{A(n) \in \Theta(\phi^n)}$$

Exponential order of growth.

7/3/17

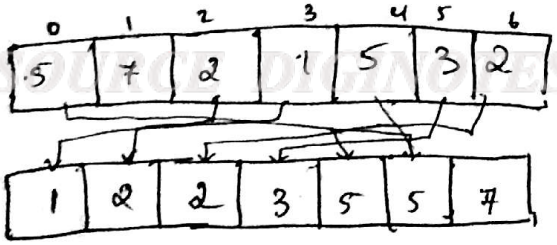
Problem types (Refer text book)

1) Sorting
ordering of elements based required manner.
Ex:- Bubble sort, merge sat, selection sort, Radix sort, Insertion sort.

2→ properties of sorting algorithm.
1. Stable property. [maintains first come first ser
2. In-place property.

1. stable property :-
A sorting Algorithm is said to be stable if it maintains the relative order of the duplicate elements even after sorting.

example



2. In-place property :-
An Alg sorting Algorith is said to be in-place if the algorithm does not consume extra memory
Ex:- Bubble sort, selection sort, Insertion sort

→ merge sort is sorting by distributed counting

2. Searching Problem
Find an element in the given list is known as Searching problem.

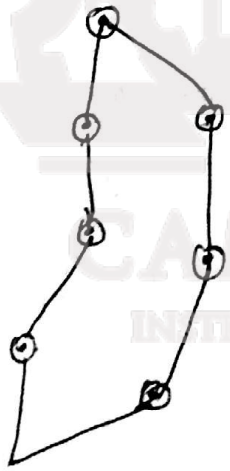
Numeric Searching
↳ simple/set of number (pattern)

Non-Numeric
↳ characters / sub-string.

Ex:- linear, binary, interpolation search, Hashing
Horspool, Boyer-more.

3. String Processing
ex:- Adding 2 string
Finding length of str.
Copying 2-strings.
Searching sub-strings.

4. Graph Problems



Shortest path.
Hamiltonian circuit.
Traversals Techniques.
Travelling sales person Problems (TSP)
Distance algorithm.
Spanning tree graph.

5) Geometric Problems
problems regarding plotting all geometric shapes

6) Numerical problems
→ equations which are continous in nature.
→ Definite integral.
↓
→ sine series,
→ Runge kutta method
→ simpsons method

7) Combinatorial problem :-

Permutation, combination, sub set which grows exponentially

Data structure

location
- Linear :- Array, stack, Queue, structure, List
- Non-Linear :- Tree, Graph.

Based on access
- sequential - Linked list, stack, queue, tree
- Dynamic - array, indexed-Linked list.

Array :- homogenous collection of objects.

Structure :- Homogenous or heterogenous collection of object.

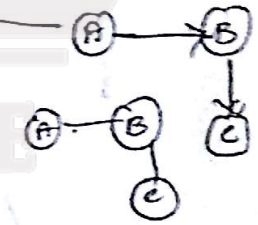
stack :- FILO, LIFO

Queue :- FIFO, LIFO

graph :- G{V, E}
- Directed
- Undirected.

1. Adjucancy matrix → n x n Symmetrix matrix.

2) Adjacency list



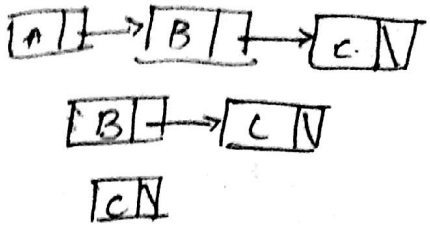
Assymmetric

	A	B	C
A	0	1	0
B	0	0	1
C	0	0	0

Symmetric

	A	B	C
A	0	1	0
B	1	0	1
C	0	1	0

Adjacency list



Sparse Graph

-> no of edges are less than no of nodes

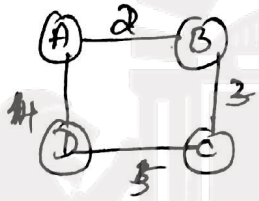
-> Graph containing few edges. [Adjacency list]

Dense Graph

-> opposite of sparse graph [Adjacency matrix]

1. weighted graph -> edge with value.

ex:- Distance blw two cities.



weight matrix or cost matrix

	A	B	C	D
A	∞	2	∞	4
B	2	∞	3	∞
C	∞	3	∞	5
D	4	∞	5	∞

Adju

connected graph.

comple graph

cylic graph

Acyclic graph

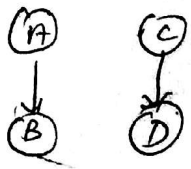
(DAG).

Tree (CDAG)

1) connected graph
2) Acyclic graph
3) Directed graph

Forest

(Not connected DAG)



indegree

Outdegree

vertex - indegree-0-root.
- outdegree - 0 - leaf.

Vertex - non-zero indegree
and non-zero outdegree. } Branch.

→ Heap tree → descending order heap tree
descending → " →

BST.

Set :- collection of non-duplicate elements

multiset → collection of elements with duplicate value.

Dictionary pair of elements
<name, value>

General method
Binary method
Merge sort
Quick sort
Min-max algorithm
Strassen's matrix multiplication.