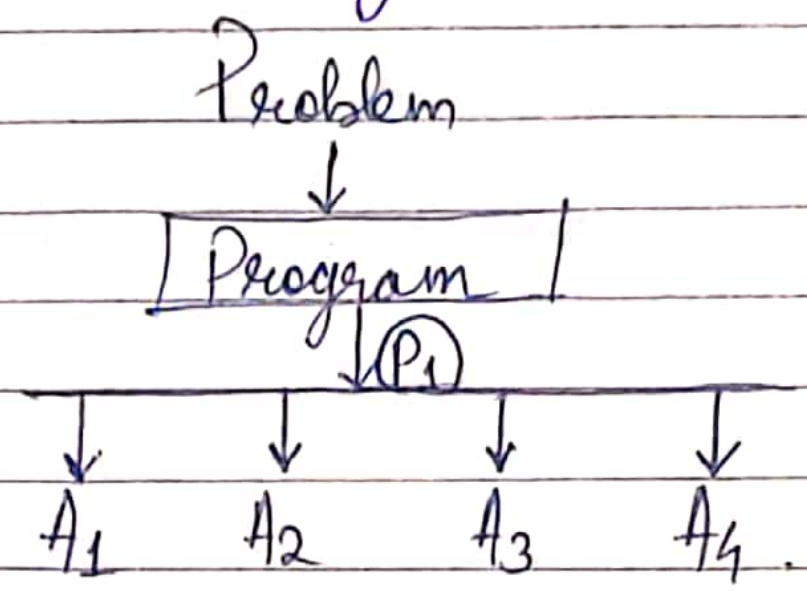
13.03.2020

(i) ⭐ Algorithms :- Informally, an algorithm is any well-defined Computational procedure that takes some value, (or) set of values, as input and produces some value, (or) Set of values, as Output.
⇒ An algorithm is thus a sequence of Computational Steps that transform the input into the Output.

(ii) Inorder to Solve any problem in Computer Science generally we write program.

(iii) Before write any program in Programming language we can write the informal description of the Sol" which is an algorithm.



(iv) If we direct write a program in Some programming language (like "C" we should write blue print. (we should write any informal language and any english like language).

(v) Let there is a problem "P_1" and "P_1" has many Sol" each Sol" has an algorithm.

(vi) "P_1" might be many Sol" like A_1, A_2, A_3, A_4

(vii) Before implementing any algorithm. any program

we need time and space. we analise the algori-m in terms of time and space.

(viii) The Course design and analysis of algorithm we will talk about how to design various algorithm for a given Problem and how to analyse them according to the analysis which will take less time and less memory.

⊛ "Criteria of any Algorithm":-
No algorithm must satisfy the following Criteria.

1.) 'Input':- 0 (or) more quantity are externally Supply.

2.) 'Output':- Atleast. One Output is Produced.

3.) "finite ness":- If we track Out the instructions of an algorithm then for all Cases the algorithm terminates after a finite no-of Steps.

4.) "definiteness":- Each Instruction is clear and unambigious.

5.) "Effectiveness":- Each and Every instruction must be very basic so that it can be Carried Out in principle by a person using Only pencil and Paper.

⊛ Correction of Algorithm :-
for any algorithm there we must prove that it always returns the desired output for all legal instruction of the problem.

⊛ Designing an Algorithm :-

$$P(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

Given $a_0, a_1, \ldots, a_n$ and x

Algorithm :- Input $a_0, a_1, a_2 \ldots, a_n$ and x

```
Set S ← a0
for i = 1 ton do
    S = S + a_i * x^i
return S.
```

Total no. of addition = n.

Total no. of multiplication $= 1+2+3+\cdots+n$

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

Since $n < \frac{n^2+n}{2} = \frac{n^2+n}{2} = O(n^2)$. ($\because$ Order of n^2)

⊛ Performance of Programs :

The performance of a program is the amount of Computer memory and time needed to run a program. we use two approaches to determine the performance of a program.

*) Analytical and *) Experimental.

*) Analytical :- In preformance / Performance analysis we use Analytical method.

*) Experimental :- In performance measurement we Conduct Experimental method.

Time Complexity:

⇒ The time needed by an algorithm expressed as a function of the size of a problem is called the time complexity of the algorithm.

⇒ The time complexity of a program is the amount of computer time it needs to run to completion.

⇒ The limiting behaviour of the complexity as size increased is called asymptotic time complexity. It is the asymptotic complexity of an algorithm, which ultimately determines the size of problems that can be solved by the algorithm.

Space Complexity:

The space complexity of a program is the amount of memory it needs to run to completion.

The space need by a program has the following components.

*) Instruction Space :- Instruction space is the space needed to store the compiled version of the program instructions.

*) Data Space :- Data space is the space needed to store all constant and variable values.

Data space has two components:

•) Space needed by constants and simple variables in program.

•) Space needed by dynamically allocated objects such as arrays and class instances.

④ Environment Stack Space :- The environment Stack is used to Save information needed to resume execution of partially Completed functions.

④ Instruction Space :- The amount of instruction Space that is needed deepends on factors Such as :

- ⇒ The Compiler used to Complete the program into machine Code.
- ⇒ The Compiler options in effect at the time of Compilation.
- ⇒ The target Computer.

⑤ Algorithm Design Goals.

The Three basic design goals that one should strive for in program are :

1. Try to Save Time
2. Try to Save Space
3. Try to Save face.

A program that runs faster is a better program, So Saving time is an Obvious goal. like wise, a program that Saves Space Over a Competing program is Considered desirable. We want to "Save face" by preventing the program from Locking up (or) generating reams of garbled data.

⊛ Classification of Algorithms

If 'n' is the number of data items to be processed (or) degree of polynomial (or) the size of the file to be sorted or the number of nodes in a graph etc.

*) [1] :- Most instructions of most programs are executed once (or) at most only a few times. If all the instructions of a program have this property we say that its running time is a constant.

*) [Logn] :- when the running time of a program is logarithmic, the program gets slightly slower as n grows. This running time commonly occures in Programs that Solve a big problem by transforming it into a smaller problem, cutting the size by some constant fraction., when n is a million, logn is a doubled. whenever n doubles, logn inc. by a constant, but logn does not double until 'n' increases to n^2.

*) [n] :- when the running time of a program is linear, it is generally the case that a small amount of processing is done on each input element. This is the optimal situation for an algorithm that must process n inputs.

*) [n.logn] :- This running time arises for algorithm that Solve a problem by breaking it up into smaller sub-problems, Solving them independently,

and then Combining the sol^n's. when n doubles, the running time more than doubles.

*) [n^2] :- when the running time of an algorithm is quadratic, it is practical for use only on relatively small problems. Quadratic running times typically arise in algorithms that process all pairs of data items (perhaps in a double nested loop) whenever n doubles, the running time increases four fold.

*) [n^3] :- Similarly, an algorithm that process triples of data items (perhaps in a triple-nested loop) has a Cubic running time and is practical for use only on small problems. whenever n doubles, the running time increases eight fold.

*) [2^n] :- few algorithms with exponential running time are likely to be appropriate for practical use, Such algorithms arise naturally as "brute-force" solutions to problems. when n doubles, the running time Squares.

Complexity of Algorithms

The Complexity of an algorithm M is the function f(n) which gives the running time and/or storage space requirement of the algorithm in terms of the size "n" of the input data. Mostly, the storage space required by an algorithm is simply a multiple of the data size "n". Complexity shall refer to the running time of the algorithm.

The function f(n), gives the running time of an algorithm, depends not only on the size 'n' of the input data but also on the particular data. The Complexity function f(n) for certain cases are:

1.) Best Case :- The Best Case Complexity of an algorithm is the function defined by the minimum number of steps at any instant size of n.

2.) Average Case :- If the function defined by the average number of step at any instant size of n

3.) Worst Case :- The worst case Complexity of an algorithm is the function defined by the maximum number of step taken at any instant size of n.

⇒ The field of Computer Science, which studies efficiency of algorithms, is known as analysis of algorithms.

=) Algorithms can be evaluated by a variety of criteria. Most often we shall be interested in the rate of growth of the time (or) space required to solve large and large instance of a problem. We will associate with the problem an integer, called the size of the problem, which is a measure of the quantity of input data.

⋆ Rate of Growth

The following notations are commonly use notations in performance analysis and used to characterize the complexity of an algorithm:

1.) Big-OH (O);
2.) Big-OMEGA (Ω);
3.) Big-THETA (Θ) and
4.) Little-OH (o).

1.) Big-OH (O) Notation:-

$f(n) \leq C \cdot g(n)$ where $n \geq n_0$.

C and n are Constant.

$C > 0$, $n_0 \geq 1$.

$f(n) = O(g(n))$ Here $f(n)$ is smaller than $g(n)$

$f(n) < g(n)$.

Big-OH notation gives an upper bound of a function for all value of n to write of n_0, the value of $f(n)$ is on (or) below of $g(n)$.

⊛ Example→1 find the Big-oh notation of the function for $f(n) = 5n^3+n^2+3n+2$

(Ans) :- $f(n) = 5n^3+n^2+3n+2$

for $n \geq 2$

$\Rightarrow 5n^3+n^2+3n+2 \leq 5n^3+n^2+3n+n$

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

for $n^2 \geq 4n$.

$\Rightarrow 5n^3+n^2+4n \leq 5n^3+n^2+4n^2$

$\leq 5n^3+2n^2$

for $n^3 \geq 2n^2$

$\Rightarrow 5n^3+2n^2 \leq 5n^3+n^3 \leq 6n^3$

for $n, n_0 \geq 2$

$C = 6$, So $g(n) = O(n^3)$

⊛ Example→2 find Big-oh notation of function of $f(n) = 2^n+6n^2+3n$

(Ans) :- 2^n+6n^2+3n

for $n^2 \geq 3n$

$\Rightarrow 2^n+6n^2+3n \leq 2^n+6n^2+n^2 \leq 2^n+7n^2$

for $2^n \geq n^2$

$\Rightarrow 2^n+7n^2 \leq 2^n+7.2^n \leq 8.2^n$

$C = 8$.

$g(n) = O(2^n)$. (Ans).

*) Theorem :- 1

Let $f(n)$ and $g(n)$ be two function, such that

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = c < \infty$$, then $f(n) = Og(n)$

Proof :- If $f(n) = Og(n)$, then $\forall$ all $n \geq n_0$ and 'c' is the positive Constant

$$f(n) \leq c \cdot g(n).$$

$$\Rightarrow \frac{f(n)}{g(n)} \leq c \quad \Rightarrow \boxed{\lim_{n \to \infty} \frac{f(n)}{g(n)} = c}$$

(*) Example :1 find the Big-oH notation for the function

$f(n) = 5n^3 + n^2 + 3n + 2$.

!:- Assume that $g(n) = O(n^3)$.

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

$$\Rightarrow \lim_{n \to \infty} \left(\frac{5n^3 + n^2 + 3n + 2}{n^3} \right)$$

$$\Rightarrow \lim_{n \to \infty} \frac{5n^3}{n^3} + \frac{n^2}{n^3} + \frac{3n}{n^3} + \frac{2}{n^3}$$

$$\Rightarrow \lim_{n \to \infty} 5 + \frac{1}{n} + \frac{3}{n^2} + \frac{2}{n^3}$$

$$\Rightarrow \lim_{n \to \infty} 5 + \frac{1}{\infty} + \frac{3}{\infty} + \frac{2}{\infty}$$

$$= 5$$

$$\therefore c = 5.$$

2. OMEGA Notation.

It is a lower bound to a function.

$$f(n) \geq c \cdot g(n)$$

$n \geq n_0$

$c > 0, \; n \geq 1$

* Example → 1 $f(n) = 5n^3 + n^2 + 3n + 2$

$\Rightarrow f(n) = 5n^3 + n^2 + 3n + 2$

for $n \geq 2$.

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

for $n^2 \geq 4n$

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

for $2n^2 \geq n^3$

$\rightarrow 5n^3 + 2n^2 \geq 5n^3 + n^3 \geq 6n^3$

So $c = 6$; $g(n) = \Omega(n^3)$.

* Example → 2 find the Omega Notation for the fun.

$f(n) = 27n^2 + 16n + 25$

(Ans) $\Rightarrow f(n) = 27n^2 + 16n + 25$

for $n \geq 25$

$\rightarrow 27n^2 + 16n + 25 \geq 27n^2 + 17n$.

for $n^2 \geq 17n$

$\rightarrow 27n^2 + 17n \geq 28n^2$.

$c = 28$. ; $g(n) = \Omega(n^2)$

3) Theta Notation:-

It is known as Sandwich Notation.

$f(n) = \theta(n)$.

where, $c_1 g(n) \leq f(n) \leq c_2 \cdot g(n)$

Here $c_1, c_2 > 0$

$n_1 \geq n_0 \geq 1$.

(*) Example-1. find the theta notation for the function.

$f(n) = 27n^2 + 16n + 25$

(Ans):- $f(n) = 27n^2 + 16n + 25$.

The given polynomial is of Order 2 i.e; $m = 2$.

$f(n) = \theta(n^3) \Rightarrow f(n) = \theta(n^2)$.

(*) Example-2 find the theta notation for $f(n) = 2^n + 6n^2 + 3n$

find the θ notation and find c_1 and c_2.

(Ans):- $f(n) = 2^n + 6n^2 + 3n$

for $n^2 \geq 3n$

$\Rightarrow 2^n + 6n^2 + 3n \geq 2^n + 6n^2 + n^2 \geq 2^n + 7n^2$.

for $2^n \geq 7n^2$

$\Rightarrow 2^n + 7n^2 \geq 2^n + 2^n \geq 2 \cdot 2^n$.

Now, $c_1 g(n) \leq f(n) \leq c_2 g(n)$.

$\Rightarrow 2^n \leq 2^n + 6n^2 + 3n \leq 8 \cdot 2^n$.

$\boxed{c = 1}$: Co-efficient of $g(n)$

= i.e 2^n is (1).

15.03.2020

Recurrence Relations

=> Recurrence Relation for a sequence of numbers is a formula that relates all but a finite number of terms of the sequence, namely, $\{a_0, a_1, a_2 \ldots, a_{n-1}\}$, for all integers n with $n \geq n_0$, where n_0 is a nonnegative integer.
Recurrence relations are also called difference equations.

=> Sequences are often most easily defined with a recurrence relation, however the calculation of terms by directly applying a recurrence relation can be time consuming. The process of determine a closed form expression for the terms of a sequence from its recurrence relation is called solving the relation.
Some guess and check with respect to solving recurrence relation are as follows:

=> Make simplifying assumptions about input

=> Tabulate the first few values of the recurrence

=> look for patterns, guess a soln.

=> Generalize the result to remove the assumptions.

Examples: factorial, Quick sort, Binary Search, Fibonnaci etc...

Recurrence relation is an equation, which is defined in terms of itself. There is no single technique (or) algorithm that can be used to solve all recurrence relations. In fact, some recurrence relations can't be solved. Most of the recurrence relations that we encounter are linear recurrence with Const. Coefficients.

⇒ Several technique like Substitution, Recursion-tree, characteristic root and generating function are available to solve recurrence relation.

(*) The Iterative Substitution method.

⇒ One way to solve a Divide-And-Conqure recurrence equation is to use the Iterative substitution method. (Plug-and-chug)

a) In using this method, we assume that the problem size "n" is fairly large and we then substitute the general form of the recurrence for each occurence of the function T on the right-hand side. for example, performing such a substitution with the merge sort recurrence equation yields the Eqn.

$$T(n) = 2(2T(n/2^2) + b(n/2)) + bn.$$
$$= 2^2 T(n/2^2) + 2bn$$

Plugging the general eqn for T gain yields the Eqn.

$$T(n) = 2^2(2T(n/2^3) + b(n/2^2)) + 2bn.$$
$$= 2^3 T(n/2^3) + 3bn.$$

The hope in applying the iterative substitution method is that, at most point, we will see a -

pattern that can be converted into a general closed-form eqn: (with T only appearing on the left-hand side). In the case of merge-sort recurrence Eqn, the general form is:

$$T(n) = 2^i T(n/2^i) + ibn.$$

Note that the general form of this Eqn shifts to the base case, $T(n) = b$, where $n = 2^i$, that is, when $i = \log n$, which implies:

$$T(n) = bn + bn \log n.$$

In other words, $T(n)$ is $O(n \log n)$.

In a general appn. of the iterative substitution technique, we hope that we can determine a general pattern for $T(n)$ and we can also figure out when the general form of $T(n)$ shifts to the base case.

⊛ Example→1 $Tn = \begin{cases} 2T(n/2) ; n>0 \\ 0 \quad \text{for } n=0 \end{cases}$ Prove that $T(n) = O(n \log n)$ by substitution method.

Soln ⇒ $T(n) = O(n \log n)$.

$T(n) ; O(n \log n)$

⇒ $T(n) \leq c \cdot n \log n$ [By defination $f(n) = O\,g(n)$ ⇒ $f(n) \leq c \cdot g(n)$]

⇒ $T(n/2) \leq c \cdot n/2 \log \frac{n}{2}$ — (1).

Given that for $n > 0$,

$\Rightarrow T(n) = 2T(n/2)$

$\Rightarrow T(n) \leq 2 \cdot c \cdot \frac{n}{2} \log \frac{n}{2}$ [$\because$ Putting the value $T(n/2)$ from eqn 1]

$\Rightarrow T(n) \leq c \cdot n \cdot \log \frac{n}{2}$.

$\Rightarrow T(n) \leq cn[\log n - \log 2]$ [$\because \log \frac{a}{b} = \log a - \log b$

$\Rightarrow T(n) \leq cn[\log n - 1]$

$\Rightarrow T(n) \leq cn \log n - cn$ [$\because cn$ is neglected - for small value]

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

So, $f(n) \leq c \cdot g(n)$

Hence, $T(n) = O(n \log n)$. ($\because$ Proved).

Example → 2 $T(n) = 2 + (n-1) + 1$.
Prove that $T(n) = O(2^n)$ by Substitution

Sol^n:- Assume that $T(n) = O(2^n)$.

$\Rightarrow T(n) \leq c \cdot 2^n$ [$\because$ By def^n $f(n) = O(g(n))$]

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

$\Rightarrow T(n-1) \leq c \cdot 2^{n-1}$

Given that

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

$T(n) = c \cdot 2T(n-1) + 1$

$T(n) \leq 2 \cdot c \cdot 2^{n-1} + 1$ [$\because$ Putting the value of $T(n-1)$ from eqn 1].

$\Rightarrow T(n) \leq 2.c.2^n.2^{-1} + 1$.

$\Rightarrow T(n) \leq c.2^n + 1$.

(∵ neglecting 1).

$\Rightarrow T(n) \leq c.2^n$

∴ Hence $T(n) = O(2^n)$ Proved

(*) The Recursion Tree:

Another way of characterizing recurrence Eq^n s is to use the recursion tree method.
Like the iterative substitution method, this technique uses repeated substitution to solve a recurrence equation, but it differs from the iterative substitution method in that, rather than being an algebraic approach, it is a visual approach.

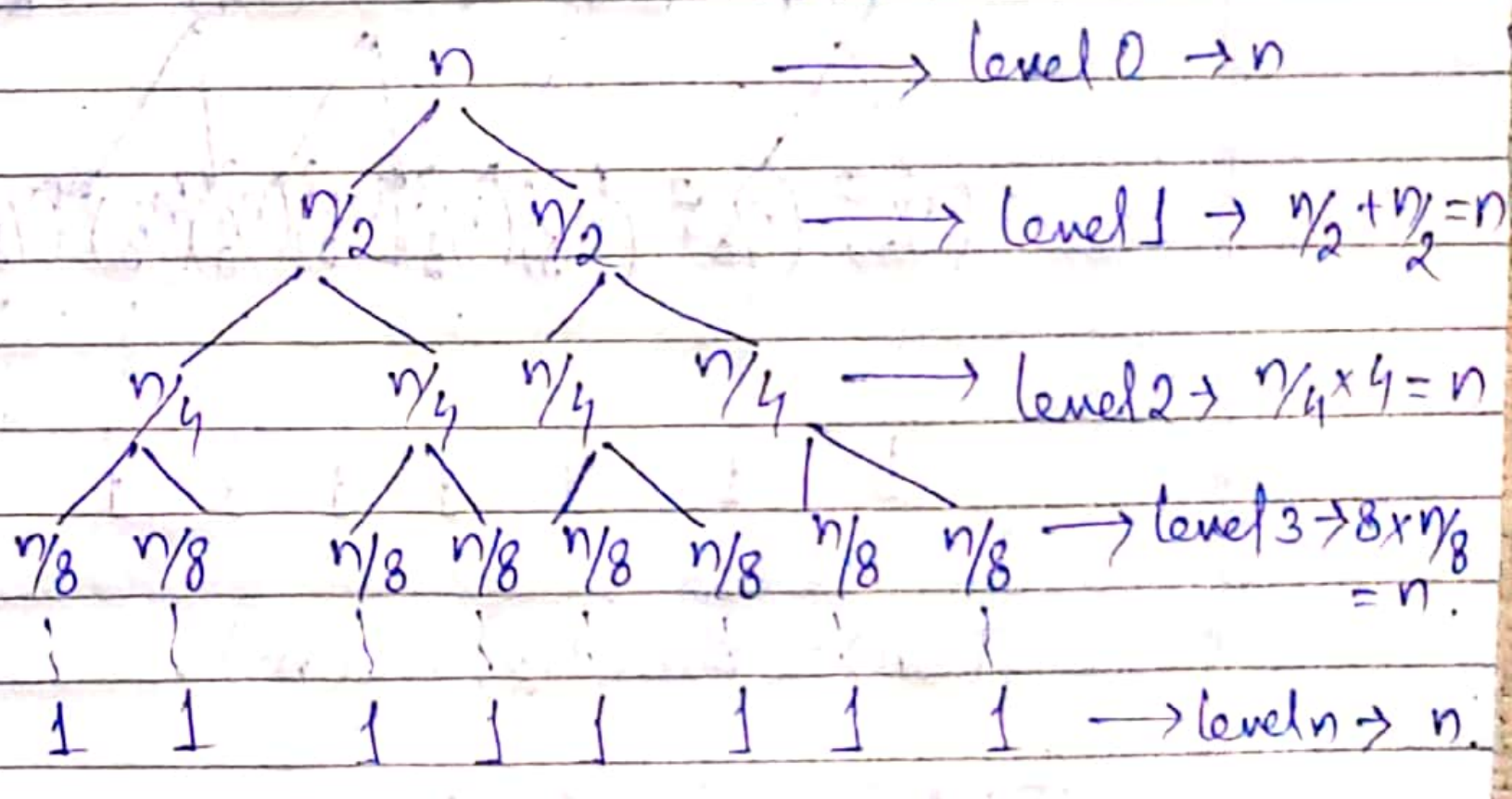
In using the recursion tree method, we draw a tree R wherein each node represents a different substitution of the recurrence eq^n. Thus, each node in R has a value of the argument n of the function $T(n)$ associated with it. In addition, we associated an Overhead with each node V in R, defined as the value of the non-recursive part of the recurrence eq^n for V.
For divide-and-Conquer recurrences, the Overhead corresponds to the running time needed to merge the subproblem solutions coming from the children of v. The recurrence eq^n is then solved by

Summing the Overheads associated with all the nodes of R. This is Commonly done by first Summing Values accross the levels of R and then Summing up these partial Sums for all the levels of R.

(*) Example(1):- Solve $T(n) = 2T(n/2) + O(n)$ by using recurssion tree method.

(Ans):- $T(n) = 2T(n/2) + O(n)$.

$= T(n/2) + T(n/2) + O(n)$.



Total time = n+n+n+---- upto logn times

= nlogn.

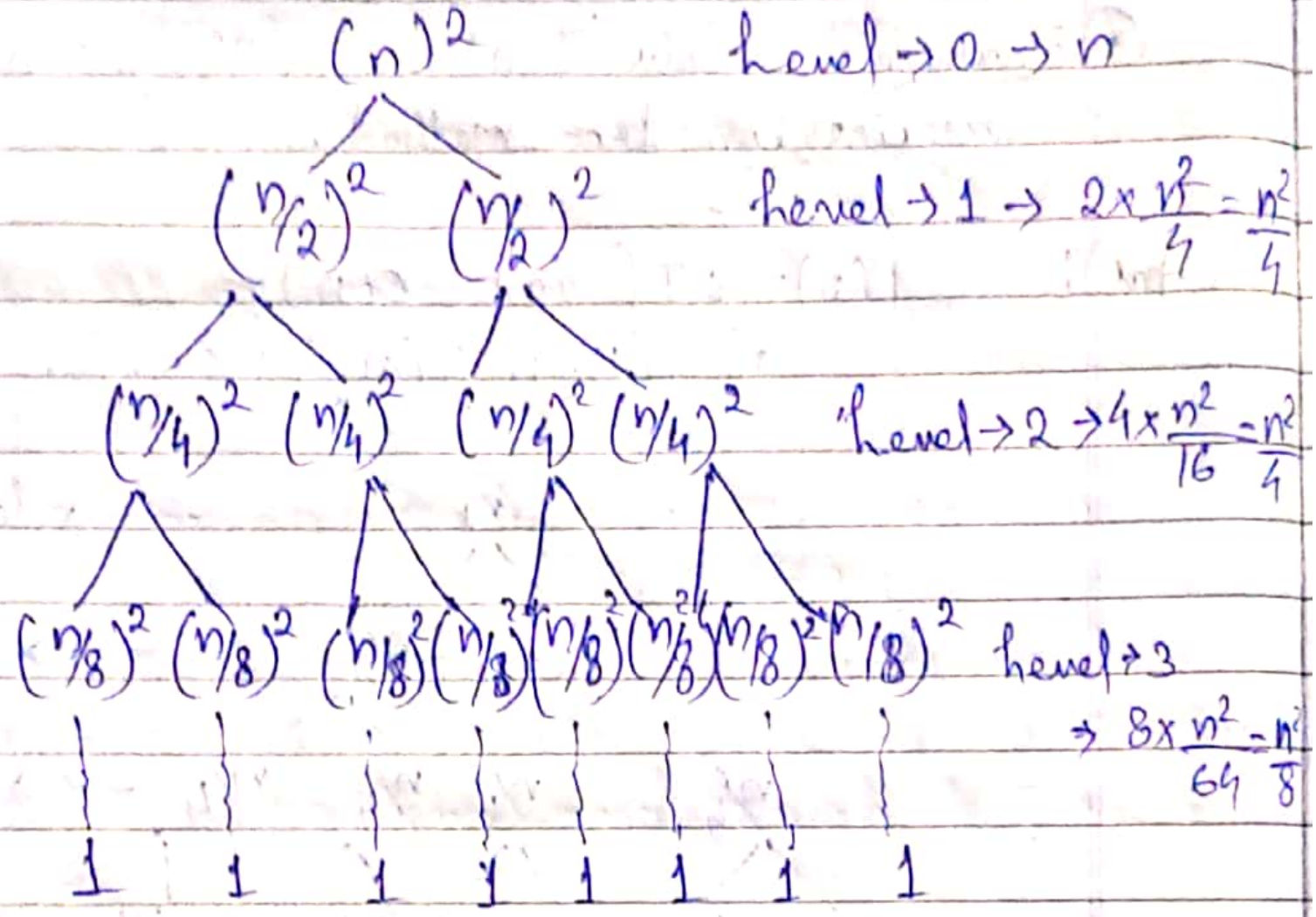
Cost of level = Addition of node present in that level.

Total Cost = Addition of all level.

(*) Example → 2 $2T(n/2) + O(n^2)$ Design the recurrsion tree and find the total time Complexity.

Soln :- $T(n) = 2T(n/2) + O(n^2)$

$\rightarrow T(n) = T(n/2) + T(n/2) + O(n^2)$.



$T(n) = 2T(n/2) + O(n^2)$.

Total time $= n^2 + \frac{n^2}{2} + \frac{n^2}{4} + \frac{n^2}{8} \ldots$

$= n^2\left(1 + 1/2 + 1/4 + 1/8 + \ldots\right)$

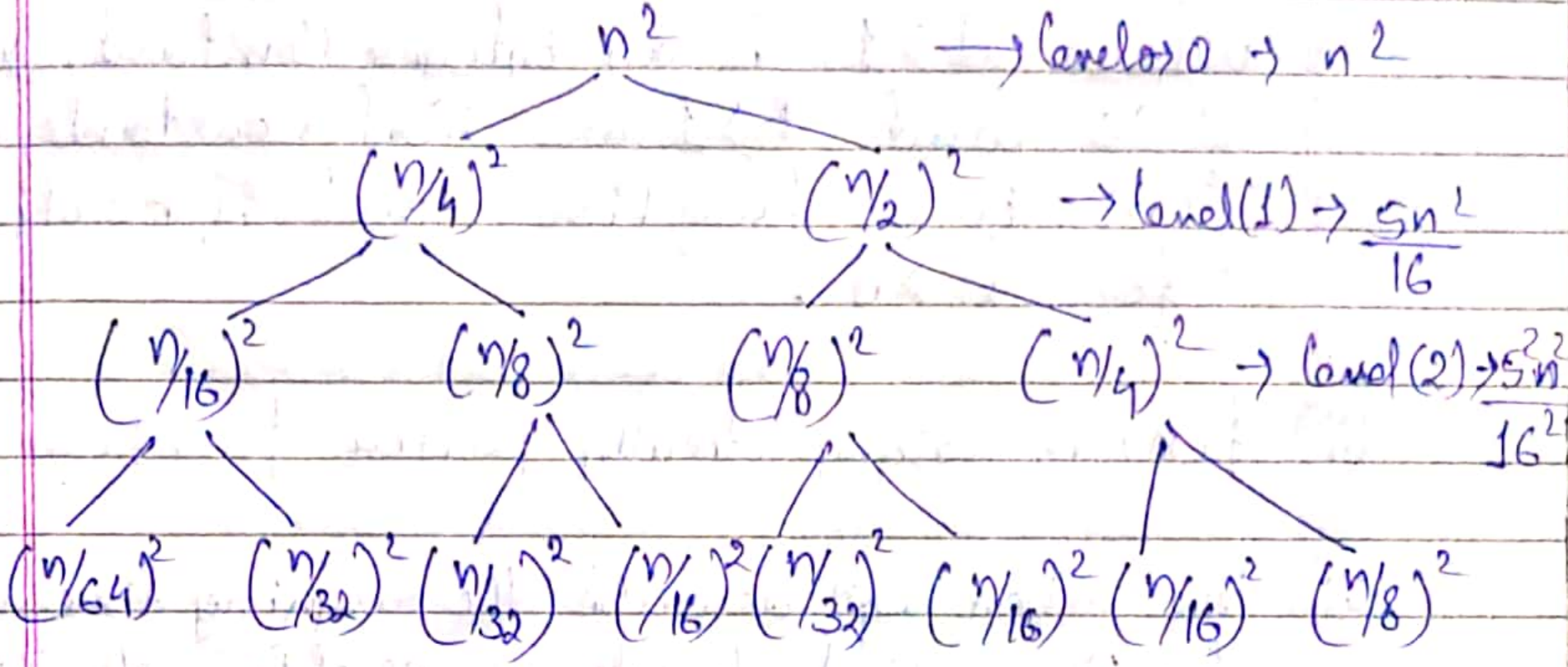
$= n^2$ (Constant) $= n^2$.

$T(n) = O(n^2)$.

* Example → 3 $T(n/4) + T(n/2) + O(n^2)$.

Design the recursion tree find the time Complexity.

(Ans):- $T(n) = T(n/4) + T(n/2) + O(n^2)$



Total time $= n^2\left(1 + \frac{5}{16} + \frac{5^2}{16^2} + \frac{5^3}{16^3}\right)$

$= n^2$

⊛ Master Method.

(i) If Solve the recurrence of the form

$$T(n) = \begin{cases} c & \text{if } n < d \\ a\,T(n/b) + f(n) & \text{if } n \geq d. \end{cases}$$

where $d > 1$ is an integer Constant, $a > 0$, $c > 0$ and $b > 1$ are real Constants, and $f(n)$ is a function that is positive for $n \geq d$.

(ii) $f(n)$ is asymptotically positive function.

(iii) The above eqn describe the running time of an algorithm that devide a problem of size n into Sub problem and each of size n/b.

(iv) $f(n)$ is the Cost of deviding and Combining the result of Sub-problem.

$$f(n) = D(N) + C(N).$$

⊛ "Master - Theorem"

The master method for Solving Such recurrence equations involves Simply writing down the answer based on whether One of the three Cases applies. Each Case is distinguished by Comparing $f(n)$ to the Special function $n^{\log_b a}$

*) The Master theorem: Let $f(n)$ and $T(n)$ be defined as above.

1.) If there is a Small Constant $\epsilon > 0$, such that $f(n)$ is $O(n^{\log_b a - \epsilon})$, then $T(n) = \theta(n^{\log_b a})$.

2.) If there is a Constant $K \geq 0$, such that $f(n)$ is $\theta(n^{\log_b a} \log^K n)$, then $T(n)$ is $\theta(n^{\log_b a} \log^{K+1} n)$.

3.) If there are Small Constant $\epsilon > 0$ and $\delta < 1$, Such that $f(n)$ is $\Omega(n^{\log_b a + \epsilon})$ and $af(n/b) < \delta f(n)$, for $n \geq d$, then $T(n)$ is $\theta(f(n))$.

⊛ Observation.

*) Case→1 Characterizes the situation where $f(n)$ is polynomial Smaller than the Special function, $n^{\log_b a}$, then $T(n) = O(n^{\log_b a})$.
($n^{\log_b a}$ is Larger).

*) Case→2 $f(n)$ is near to $n^{\log_b a}$ then,
$T(n) = O(n^{\log_b a} \cdot \log n)$.

*) Case→3 $f(n)$ is Larger than $n^{\log_b a}$
$T(n) = O(f(n))$.

Date 16.03.20
Page

* Example → 1 $T(n) = 4T(n/2) + n$.

(Ans) → Here $a = 4 ; b = 2 ; f(n) = n$.

$n^{\log_b a} = n^{\log_2 4} = n^2$.

Here $n^{\log_b a}$ is Larger than $f(n)$.
Hence by applying Case → 1

$T(n) = O(n^{\log_b a}) = O(n^2)$.

* Example → 2 $T(n) = 3T(n/4) + n\log n$.

(Ans) → $a = 3 ; b = 4 ; f(n) = n\log n$.

→ $n^{\log_b a} = n^{\log_4 3}$

$= n^{0.793}$

So, $T(n) = O(f(n)) = O(n\log n)$. (Ans).

* Example → 3 $T(n) = 2T(n/2) + n$.

(Ans) → $a = 2 ; b = 2 ; f(n) = n$.

→ $n^{\log_b a} = n^{\log_2 2} = n$.

Case → 2.

→ $T(n) = O(n^{\log_b a} \cdot \log n) = O(n \cdot \log n)$.

(Ans)

(*) Example → 4. $T(n) = 4T(n/2) + n^2$

Sol'n ⇒ $a = 4 ; b = 2 ; f(n) = n^2$.

$\Rightarrow n^{\log_b a} = n^{\log_2 4} = n^2$

$\Rightarrow T(n) = O(n^{\log_b a} \cdot \log n)$

$= O(n^2 \cdot \log n)$. (Ans)

(*) Example → 5 $T(n) = 4T(n/2) + n^3$

Sol'n ⇒ $a = 4 ; b = 2 ; f(n) = n^3$.

$n^{\log_b a} = n^{\log_2 4} = n^2$.

$T(n) = O(f(n)) = O(n^3)$. (Ans)

(*) Example → 6 $T(n) = 2T(n/2) + n^3$

Sol'n ⇒ $a = 2 ; b = 2 ; f(n) = n^3$.

$n^{\log_b a} = n^{\log_2 2} = n$.

$T(n) = O(f(n)) = O(n^3)$. (Ans)

(*) Example → 7 $T(n) = T(9n/10) + n$

Sol'n ⇒ $a = 1 ; b = 10/9 ; f(n) = n$

$\Rightarrow n^{\log_b a} = n^{\log_{10/9} 1} = 1$.

$T(n) = O(f(n)) = O(n)$. (Ans)

Date
Page

(*) Example → 8 $T(n) = 16T(n/4) + n^2$.

Sol^n :- $a = 16$; $b = 4$; $f(n) = n^2$.

$\Rightarrow n^{\log_b a} = n^{\log_4 16} = n^2$.

$\Rightarrow T(n) = \Theta(n^{\log_b a} \cdot \log n) = \Theta(n^2 \cdot \log n)$

(Ans)

(*) Example → 9 $T(n) = 7T(n/3) + n^2$.

Sol^n :- $a = 7$; $b = 3$; $f(n) = n^2$.

$n^{\log_b a} = n^{\log_3 7} = n^{1.77}$

$\Rightarrow$ T

(*) Example → 10 $T(n) = 7T(n/2) + n^2$.

Sol^n :- $a = 7$, $b = 2$; $f(n) = n^2$.

$\Rightarrow n^{\log_b a} = n^{\log_2 7} = n^{2.80}$

$\Rightarrow T(n) = \Theta(n^{\log_b a}) = \Theta(n^{2.80})$. (Ans)

(*). Example → 11 $T(n) = 2T(n/4) + \sqrt{n}$.

Sol^n :- $a = 2$; $b = 4$; $f(n) = \sqrt{n}$.

$\Rightarrow n^{\log_b a} = n^{\log_4 2} = n^{0.5} = \sqrt{n}$.

$\Rightarrow T(n) = \Theta(n^{\log_b a} \cdot \log n) = \Theta(\sqrt{n} \log n)$.

(*) Example :- 12. $T(n) = 2T(n/2) + \log n$.

(Soln) :- $a = 2$, $b = 2$; $f(n) = \log n$.

$\Rightarrow n^{\log_b a} = n^{\log_2 2} = n$.

(∵ Case → 3).

$\Rightarrow T(n) = O(\log n)$. (Ans)

(*) Example → 13. $T(n) = 8T(n/2) + n^2$

Soln :- $a = 8$; $b = 2$; $f(n) = n^2$.

$\Rightarrow n^{\log_b a} = n^{\log_2 8} = n^3$.

(∵ Case → 1)

$\Rightarrow T(n) = O(n^3)$. (Ans)

(*) Example → 14. $T(n) = 16T(n/2) + (n \log n)^2$.

Soln :- $a = 16$; $b = 2$; $f(n) = (n \log n)^2$.

$\Rightarrow n^{\log_b a} = n^4$.

(∵ Case → 1).

$\therefore T(n) = O(n^4)$. (Ans).

$\therefore T(n) = O(n^4)$

(*) Example :→ 15 $T(n) = 3T(n/4) + n \log n$.

Soln :- $n^{\log_b a} = n^{0.79}$ $\Rightarrow a = 3$; $b = 4$; $f(n) = n \log n$

(∵ Case → 3).

$\therefore T(n) = O(n \log n)$. (Ans)