1) Algorithm definition and properties

Techniques:-

i) Brute Force
ii) Divide & Conquer
iii) Decrease & Conquer
iv) Transform & Conquer
v) Dynamic Programming
vi) Greedy
vii) Backtracking

Algorithm definition

1) is a method to solve a problem

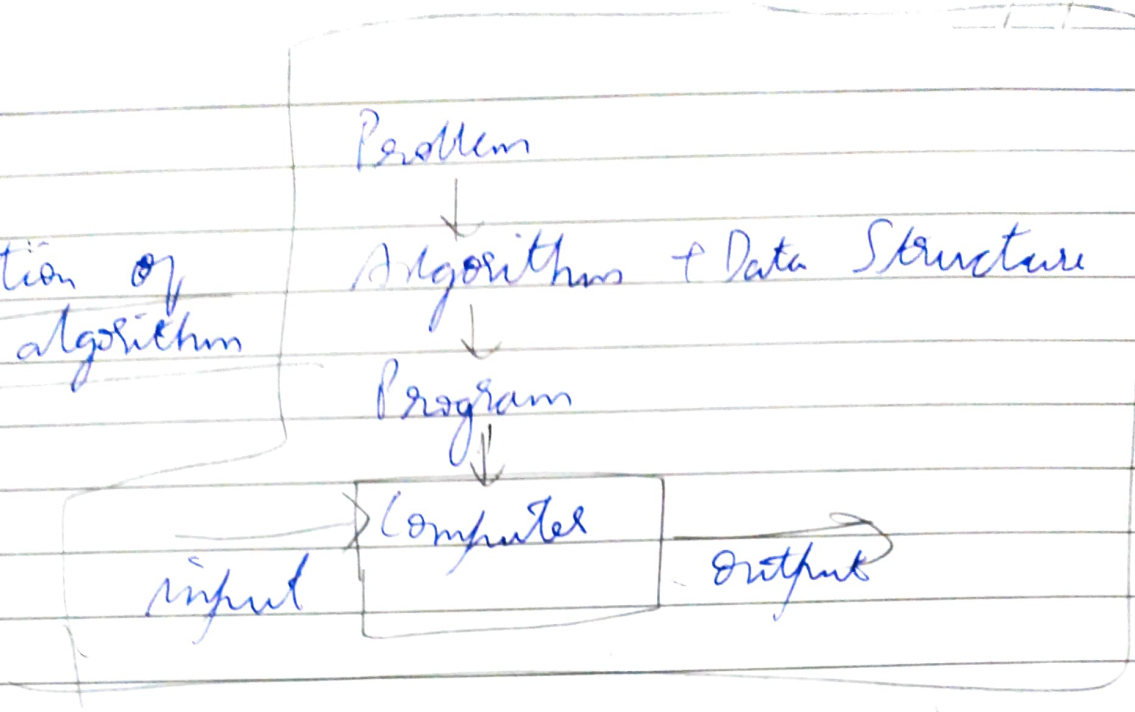
2) is a sequence of computational steps that transforms input into output (Cormen)

3) is a sequence of unambiguous instructions for solving a problem (Levitin)

4) is a finite set of instructions that if followed, accomplishes a particulas task (Schaum)

PTO

Notion of an algorithm



Properties of an algorithm

1) Input - Zero or more inputs
2) Output - One or more ~~o/p~~ outputs
3) Finiteness - Should terminate
4) Definiteness - Each instruction in an algorithm
   i) should be clear (no ambiguity)
   ii) should take finite amount of time
5) Effectiveness - If all the instructions (steps) followed properly, we should get the desired output

2) Algorithms to find GCD

Algorithm to find GCD of 2 no's

1) Euclid's Algorithm

Input: Two positive no's a and b

Output: Largest integer which divides a & b

Step 1: If b is zero, GCD is a, else go to step 2

Step 2: Find r, which is the remainder after dividing a by b

Step 3: Exchange a by b and b by r. Go to step 1

Eg:- a = 60, b = 10

a	b	r
60	10	0
10	0	4
0	4	2
4	2	0
2	0	

∴ (2) is the GCD of 60 & 10.

Pseudocode (mixture of programming language constructs with English statements)

Euclid(a, b)

// This algorithm finds GCD of 2 +ve no's
// Input : 2 no's a & b
// Output : GCD of 2 no's

while b ≠ 0 do
    r ← a mode b
    a ← b
    b ← r
return a

2) Consecutive integer checking method →

Input : a & b
Output : GCD of a & b

Step 1 : Assign t = min(a, b)

Step 2 : Perform the operation a mod t, if remainder is zero, go to step 3, else go to step 4

Step 3 : Perform b mod t, if remainder is zero GCD is t, else go to step 4

Step 4 : Decrement the value of t by 1, go to step 2

GCD $(a, b) \leq \min(a, b)$

Eg: $a = 6, b = 10, t = 6$

t	a mod t	b mod t	
6	0	4	∴ GCD(6, 10) = 2
5	1	—	
4	2	—	Problem: if either a or b is zero the algorithm will not work
3	0	1	
2	0	0	

(3) Middle school method → Input = a & b
Output = GCD of a & b

Step 1:- Find prime factors of a

Step 2:- Find prime factors of b

Step 3:- The product of common factors is the GCD

Eg:- $a = 6, b = 10$

Hence the GCD(6, 10) =

2 | 6	2 | 10
3	5

Factors of 6 = 2 × 3
Factors of 3 = 2 × 5

Drawback → It is not clearly specified how to find the prime factors (ambiguity)

↓

Definiteness / Effectiveness

Method fails when either of the input is 1

3) Efficiency of an algorithm

Algorithm efficiency
(OR)
Algorithm complexity

1) Space efficiency (space complexity)
   - is the amount of space required by an algorithm to solve the problem
   - $S(P) = C + S_p(\text{instance})$
     - C: constant part (independent of variables)
     - S_p(instance): variable part (dependent auxillary variables)

Time complexity (Time efficiency)
- indicates how fast the algorithm runs
(OR)
how long a program takes to process a given input

Space complexity

Eg (1) Algorithm sum(a, n)

// Input : 2 no's a & n
// output : Sum of a and n

return a + n

$S(P) = C + S_p$ (instance)

$C = 1 + 1 = 2$

$S_p = 0$

$\therefore S(P) = 2 + 0 = 2$

↑

Constant → O(1)

Eg (2) Algorithm Array Sum (A, n)

// Input : An array of elements of size n

// output : Sum of array elements

```
Sum ← 0
for i ← 0 to n-1 do
        sum ← sum + A[i]
return sum
```

$S(P) = C + S_p(\text{instance})$

$C = 1 + 1 = 2$

$S_p(\text{instance}) = 10n$

$S(P) = 2 + 10n = 10n = O(n)$

Time complexity

→ indicates how fast the algorithm executes

Time efficiency depends on

1) System on which its been executed
2) Programming language used
3) Compiler used

} → Actual time complexity cannot be calculated

→ To find the time complexity we calculate how many times the "basic operation" gets executed

$T(n) = C_{op} \times C(n)$ → function which express how many times the basic operation is executed

$T(n)$ ↓ Time complexity

C_{op} ↓ time taken for one execution of the basic operation

Time Complexity

Basic operation → most important operation of the algorithm, i.e the operation contributing the most of the total running time

Best case → minimum no. of times the basic operation gets executed for input n

Eg:- Linear search – $\Omega(1)$

Worst case → maximum no. of the time the basic operation gets executed for input n

Eg:- Linear search – $O(n)$

Average case →

4) Time Complexity of Algorithm

Let us assume that the basic operation takes 1 unit of time

(1) sum → a + b $TC = 1$

(2) for i ← 1 to n do → $TC = n$
    sum ← sum + A[i]

(3) for i ← 1 to n do
    for j ← 1 to n do
        sum ← sum + A[i][j]

$TC = n^2$

(4) for (i = 1; i <= n; i++)
        sum = sum + A[i]

} n

for (i = 1; i <= n; i++)
    for (j = 1; j <= n; j++)
        if (A[i] > B[j])
            swap (A[i], B[j])

} n^2

Since $n^2 > n$, $TC = n^2$

(5) for (i = 1; i <= n; i++)
    for (j = 1; j <= i; j++)
        for (k = 1; k <= 25; k++)
            (Basic operation)

$25(1 + 2 + 3 \ldots\ldots n)$

$= 25 \frac{n(n+1)}{2}$

$\Rightarrow TC = n^2$.

(6) while (n > 0)
{

(Basic operation)
$n = n/2$
}

$TC = \log_2 n$

(7) while ($n > 0$)
{
(Basic operation)
$n = n - 1$
}

$TC = n$

(8) for ($i = 1$; $i <= n$; $i++$)
for ($j = 1$; $j <= i^2$; $j++$)
for ($k = 1$; $k <= \frac{n}{2}$; $k++$)
(Basic operation)

$i \rightarrow n$
$j \rightarrow n^2$
$k \rightarrow \frac{n}{2}$

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

__/__/____

$\sum n^3 = \frac{n^2(n-1)^2}{4} \quad \lambda 2$

$= (n^4)$

9. for (i=1, i<=n, i++)
   for (j=1, j<=n, j=j+i)

i →	1	2	3	.....	n
j →	n	$\frac{n}{2}$	$\frac{n}{3}$	.	$\frac{n}{n}$

$n + n + n + \cdots$

$n + \frac{n}{2} + \frac{n}{3} + \cdots + \frac{n}{n}$

$= (n \log n)$

5. Efficiency Class

1) Constant - O(1) → Best case scenario of linear search

2) n - linear → Worst case scenario of linear search, Worst case scenario of search in a BST

__/__/____

3) n^2 – quadratic → Bubble sort, selection sort, quicksort (worstcase)

4) $\log n$ → Worst case scenario of Binary Search in a complete BST, we perform a search operation

5) $n \log n$ → mergesort, quicksort (rest)

6) n^3 → multiplication of 2 matrices

7) 2^n → exponential → n queen's problem, subset problem

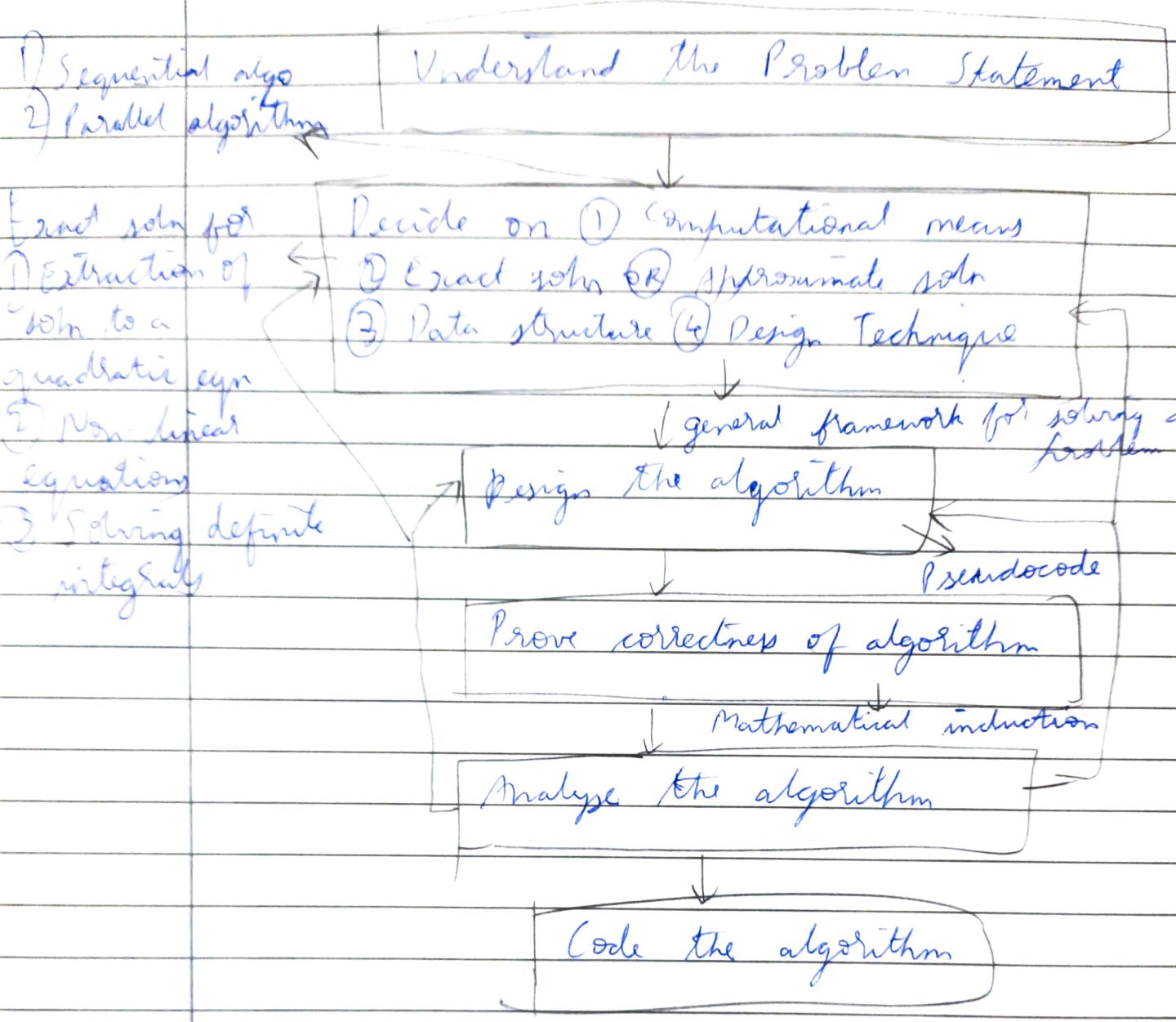
8) $n!$ → Generating all permutations to solve the problem

Constant < $\log n$ < n < $n \log n$ < n^2 < n^3 ... < 2^n < $n!$

Process of Algorithm design & analysis

PTO

__/__/____



A good algorithm is a result of effort and rework

6 Asymptotic Notation !

→ To compare and rank the order of growth of a function (Expresses the time complexity of an algorithm); we use asymptotic notation

→ There are 5 different types of asymptotic notation

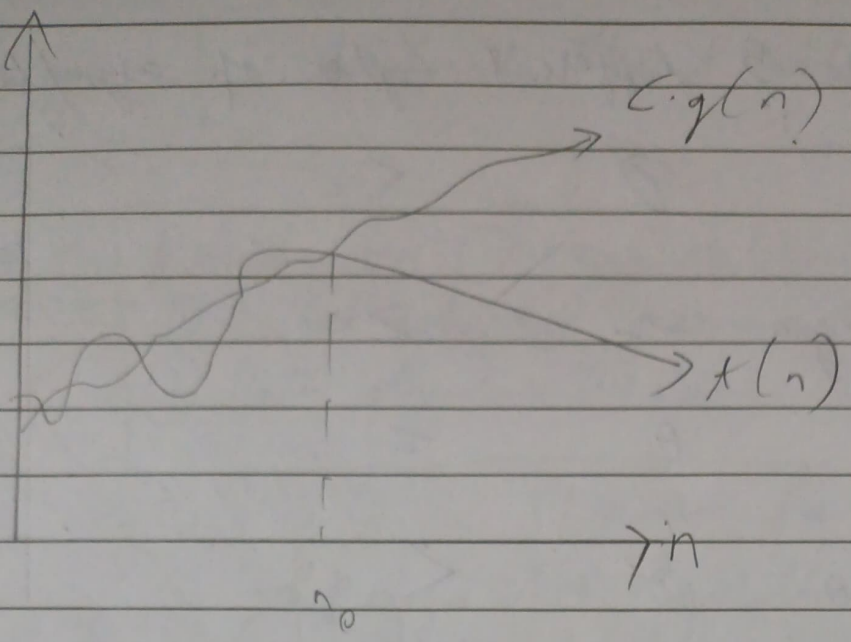
1) Big O — O — $<=$
2) Big omega — Ω — $>=$
3) Theta — θ — $=$
4) Small o — o — $<$
5) Small omega — ω — $>$

Let $f(n)$ & $g(n)$ be two non-negative functions on a set of natural no's

$f(n)$ - Function which denotes actual time taken by an algorithm
$g(n)$ - Sample function

Big O (O) Notation

→ A function $f(n)$ is said to be $O(g(n))$ denoted by $f(n) \in O(g(n))$, if $f(n)$ is bounded above some constant multiples of $g(n)$ for all larger values of n, if there exists a positive integer constant 'c' and a positive integer n_0 satisfying the statement $f(n) \leq c \cdot g(n)$ $\forall$ $n \geq n_0$.
$c > 0$, $n_0 \geq 1$



→ f(n) grows with same rate of lesser with g(n)

Eg:- $f(n) = 3n + 2$

$g(n) = n$

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

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

$3n + 2 \leq cn$

$c = 1$ $\quad \therefore n_0 = 2$

$c = 2$

$c = 3$ $\quad \therefore 3n + 2 = O(n)$

$c = 4$

$\therefore 3n + 2 \leq 4n$

$n_0 = 1$

$5 \leq 4$ ✗

$n_0 = 2$

$8 \leq 8$ ✓

__/__/__

$1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < \cdots < 2^n < 3^n < n!$

$n^2 = O(n^3)$ ✓

$n^2 = O(n^2)$ ✓

$n^3 = O(n!)$ ✓

$n^2 = O(\log^i n)$ ✗

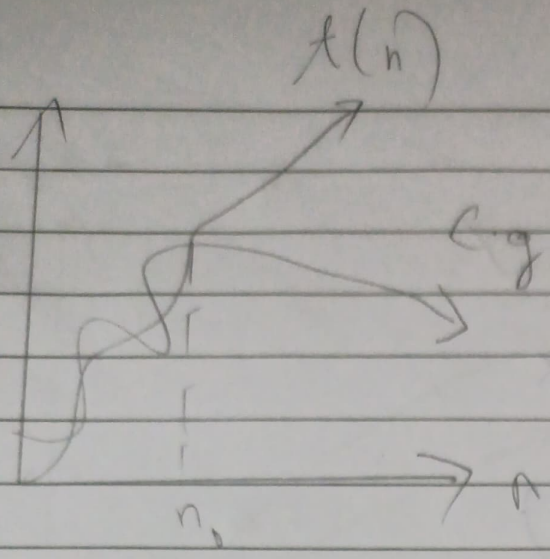
3) Asymptotic Notation - 2

Omega Notation (Ω)

→ A function $f(n)$ is said to be $\Omega(g(n))$, denoted by $f(n) \in \Omega(g(n))$, if $f(n)$ is bounded below some constant multiples of $g(n)$ for all larger values of n, if there exists a positive integer constant 'c' and a positive integer n_0 satisfying the statement

$$f(n) \geq c \cdot g(n) \quad \forall \quad n \geq n_0, \; c > 0, \; n_0 \geq 1$$

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



Eg:- $f(n) = 3n+2$

$g(n) = n$

$f(n) \geq c \cdot g(n)$ 等

$\Rightarrow 3n+2 \geq cn$

$c=1$	$3n+2 \geq 3n$
$3n+2 \geq 1n$	$\Rightarrow n=1$ ✓
$c=2$	✓
$3n+2 \geq 2n$	$3+2 \geq 3$
$c=3$	$n=2$ ✓
$3n+2 \geq 3n$	$8+2 \geq 6$
$c=4$ ✗	
$\therefore c=3$	

$1 < \log < \sqrt{n} \leq n < n\log n < n^2 < n^3 < 2^n < 3^n < n!$

$n^2 = \Omega(n^3)$ ✗

$n^3 = \Omega(n^2)$ ✓

$n! = \Omega(n^{\log n})$ ✓

$2^n = \Omega(n^2)$ ✓

$n^3 = \Omega(3^n)$ ✗

Theta Notation (Θ)

→ A function $f(n)$ is said to be $\Theta(g(n))$ denoted by $f(n) \in \Theta(g(n))$, if $f(n)$ is bounded above and below by some positive constant multiples of $g(n)$ for all large values of n if there exists some positive constants c_1 & c_2 and some non-negative integer n_0 such that

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

→ $f(n)$ can contain any function which has the same rate as $g(n)$

$f(n) = 3n + 2$

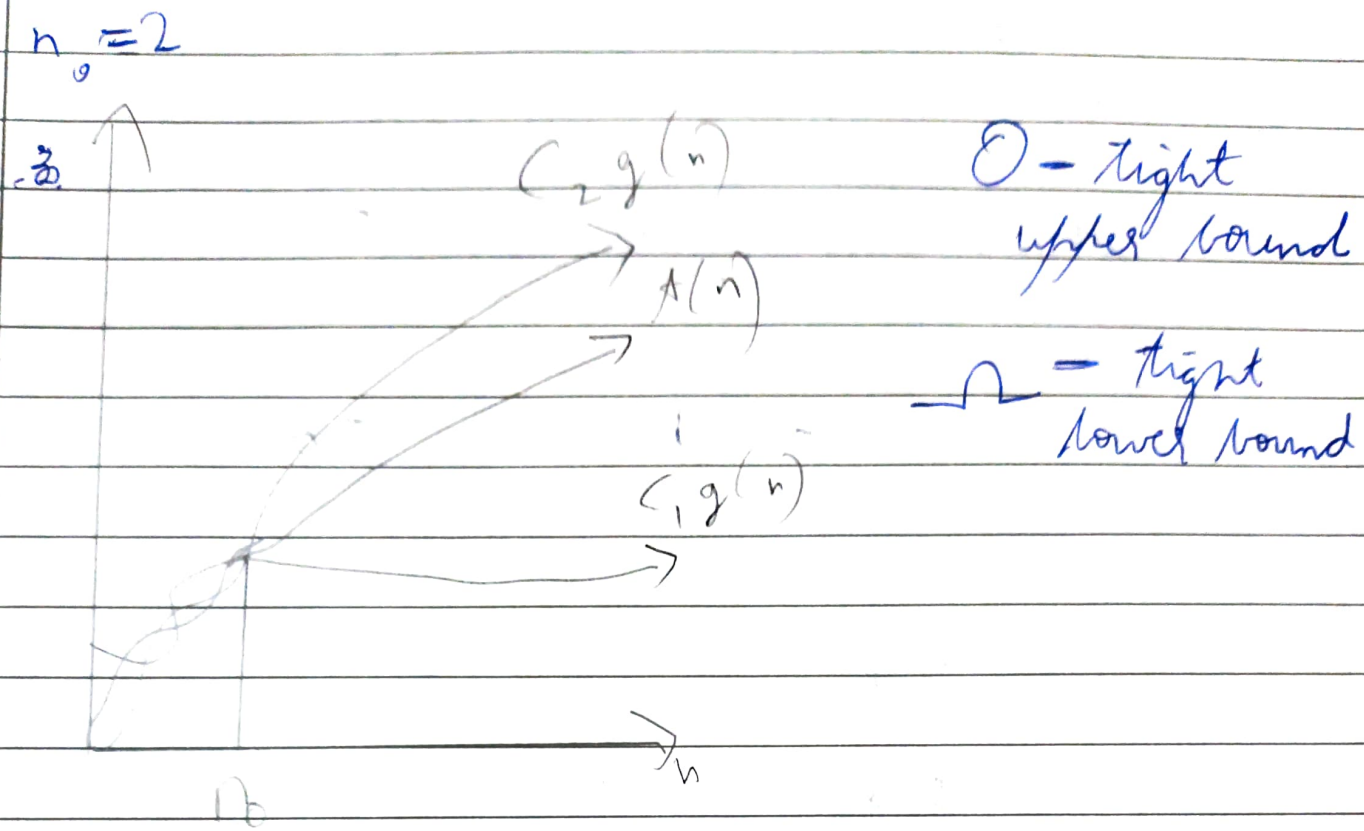
$g(n) = n$

$3n + 2 = \Theta(n)$

$c_2 n \leq 3n + 2 \leq c_1 g(n)$

$c_1 = 4 \quad n = 2$

$c_2 = 3 \quad n_0 = 1$



B) Small oh & Small omega notation

$$f(n) = o(g(n)) \text{ iff } \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

should

$g(n)$ ~~should~~ be varially bigger than $f(n)$

Eg 1) $f(n) = n$

$g(n) = n^2$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \lim_{n \to \infty} \frac{n}{n^2} = \lim_{n \to \infty} \frac{1}{n} = 0$$

$\therefore n = o(n^2)$

2) $f(n) = \log n$

$g(n) = n$

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

$$= \frac{1}{n} \lim_{n\to\infty} \frac{1}{n} = 0$$

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

Small Omega (ω) Notation

$$f(n) = \omega(g(n)) \text{ iff } \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

Eg: (1) $f(n) = 4^n$, $g(n) = 2^n$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{4^n}{2^n} = \left(\frac{4}{2}\right)^n = 2^n = 2^\infty = \infty$$

$$\therefore 4^n = \omega(2^n)$$

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

$g(n) = \log n$

Indefinite

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

$$\therefore \lim_{n\to\infty} \frac{n^2}{\log n} = \lim_{n\to\infty} 2n^2 = \infty$$

$$\therefore n^2 = \omega(\log n)$$

9 Comparison of efficiency class using asymptotic notation

$t(n) = O(g(n))$ iff $\lim_{n\to\infty} \frac{t(n)}{g(n)} \leq c$

$t(n) = \Omega(g(n))$ iff $\lim_{n\to\infty} \frac{t(n)}{g(n)} \geq c$

$t(n) = \Theta(g(n))$ iff $\lim_{n\to\infty} \frac{t(n)}{g(n)} = c$

$t(n) = o(g(n))$ iff $\lim_{n\to\infty} \frac{t(n)}{g(n)} = 0$

$t(n) = \omega(g(n))$ iff $\lim_{n\to\infty} \frac{t(n)}{g(n)} = \infty$

where c is a constant, $c > 0$.

Eg) $t(n) = n^3$, $g(n) = 3^n$

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \lim_{n\to\infty} \frac{n^3}{3^n} = \lim_{n\to\infty} \frac{3n^2}{3^n \ln 3} = c \lim_{n\to\infty} \frac{n^2}{3^n}$$

$$\leq C \lim_{n\to\infty} \frac{2n^{3n}}{3^n \ln 3} = C \lim_{n\to\infty} \frac{n}{3^n}$$

$$= C_1 \lim_{n\to\infty} \frac{1}{3^n \ln 3} + C_2 \lim_{n\to\infty} \frac{1}{3^n} = 0$$

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

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

Eg 2 $f(n) = n^{0.2}, g(n) = (\ln n)^3$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{n^{0.2}}{(\ln n)^3} = \frac{\infty}{\infty}$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{0.2 n^{-0.8}}{3 \ln^2 n \cdot n^{-1}}$$

$$= C \lim_{n\to\infty} \frac{n^{0.2}}{\ln^2 n}$$

$$= C \lim_{n\to\infty} \frac{0.2 n^{-0.2}}{2 \ln n \cdot n^{-1}}$$

$$= C \lim_{n\to\infty} \frac{n^{0.2}}{\ln n} = C_1 \lim_{n\to\infty} \frac{0.2 n^{-0.2}}{n^{-1}}$$

$$= C_1 \lim_{n\to\infty} n^{0.2} = C_2 \lim_{n\to\infty} n^{0.2} = \infty$$

b) Time Complexity of Non-Recursive algorithm

General plan

1) Decide on parameter (or parameters) indicating the input size.

2) Identify the algorithm's basic operation

3) Check whether the no. of times the basic operation executed depends only on the size of input. If it also depends on some additional property, then the worst & average case efficiency must be investigated

4) Setup a sum expressing the number of times the basic operation gets executed.

5) Using standard formula & rules of manipulation, either find a closed form for the count or at least establish the order of growth

Algorithm MaxElement(A[0...n-1])
// Finds the max element in an array
// Input: An array A[0...n-1]
// Output: Max element
maxValue ← A[0]
for i ← 1 to n-1 do
    if A[i] > maxValue
        maxValue ← A[i]
return maxValue

Input size - n
Basic operation is comparison → C

Let C(n) denotes the no. of times the comparison gets executed

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

$$= n-1-1+1 = n-1 \in n$$

2) Algorithm Element Uniqueness ($A[0 \ldots n-1]$)
// Finds whether the elements in the array are distinct or not
// Input: An array $A[0 \ldots n-1]$
// Output: Returns True if all elements are distinct, if not returns 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
```

Eg: 10 20 30 40 50 60
↳ TRUE

10 20 30 40 20 50
↳ FALSE

Input size = n
Basic operation is comparison → C — 1 unit of time

$C_{worst}(n)$ – Worst case scenario

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

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

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

$$= (n-1)+(n-2)+(n-3)+\cdots+1$$

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

11) Time Complexity of Recursive Algorithm

General plan

1) Decide on parameter (or parameters) indicating the input size

2) Identify the algorithm's basic operation

3) Check whether the number of times the basic operation executed depends only on the size of the input. If it also depends on some additional property, then the worst & average case efficiency must be investigated

4) Setup a recurrence relation with an appropriate base case (initial condition) for the no of times the basic operation gets executed

5) Solve the recurrence relation with appropriate methods to find the time complexity

↓

1) Substitution method
2) Recursive Tree method
3) Master theorem method

Algorithm Fact(n)

// To find the factorial of a no.
// Input: A positive integer n
// Output: Factorial of a given number

if n == 0 return 1
return n * Fact(n-1)

Fact(2) = 2 x 1 = 2
Fact(3) = 3 x 2 x 1 = 6
Fact(4) = 4 x 3 x 2 x 1 = 24

Input size is n
Basic operation is multiplication

↓

M

Let M(n) denotes the no. of times the basic operation gets executed

Case :- if n is 0, no multiplication is performed

$M(n)$

$M(0) = 0$

$M(n) = M(n-1) + 1 \quad \forall \; n \geq 1$

Substitution

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

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

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

$= M(n-k) = k$

$M(n-k) = 0$

$n - k = 0$

$n = k$

$\Rightarrow M(0) + k = n$

Algorithm Towel of Hanoi

// Solves the Tower of Hanoi Problem
// Input : No. of discs = n, 3 pegs S, D, I
// output : All n discs - stacks on D

If $n == 1$

move a disc from S to D

Base Else

Tower of Hanoi $(n-1, S, T, D)$
move n^{th} disc from S to D
Tower of Hanoi $(n-1, T, D, S)$

Input Size = $n \rightarrow$ No. of disc

Basic operation - Movement of disc

$\downarrow$

M

Base case: If $n=1$, We do 1 movement

$M(n)$: $\boxed{M(1)=1}$ $n=1$

$M(n) = M(n-1) + 1 + M(n-1)$ $\forall\ n>1$

$\Rightarrow M(n) = 2M(n-1) + 1$

~~$2M(n-2)$~~

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

$= 4M(n-2)+3$

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

$= 8M(n-3)+7$

$2^k M(n-k) + 2^{k-1} + 2^{k-2} + \dots 2 + 2^0$

Base case: $n-k=1$

$k = n-1$

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

$= 2^n - 1 \approx 2^n$

12 Empirical Analysis of an algorithm

13 General plan for empirical analysis of an algorithm

1) Understand the experiment's purpose

2) Decide on efficiency metrics M to be measured & measurement unit

3) Decide on the characteristics of input size (size range)

4) Prepare a program implementing the algorithm for the experimentation

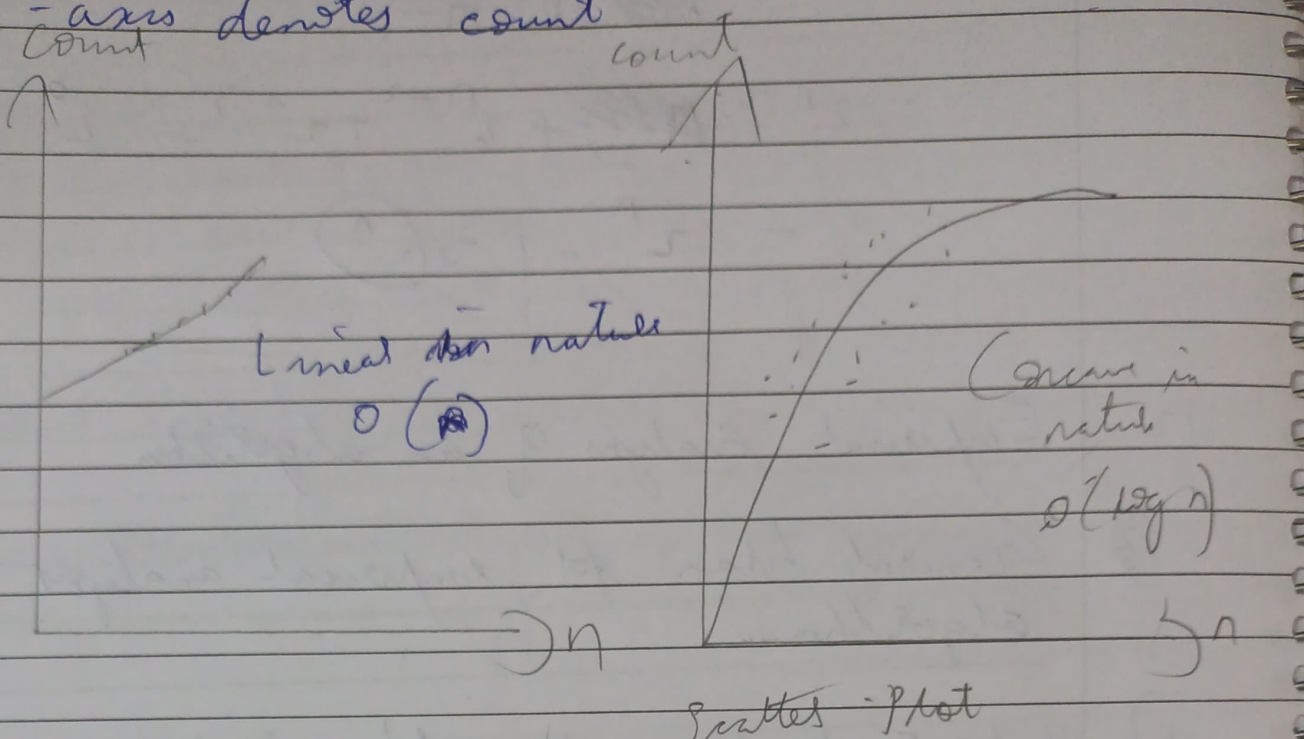
5) Generate a sample of inputs

6) Run the program on the same inputs and record the data observed

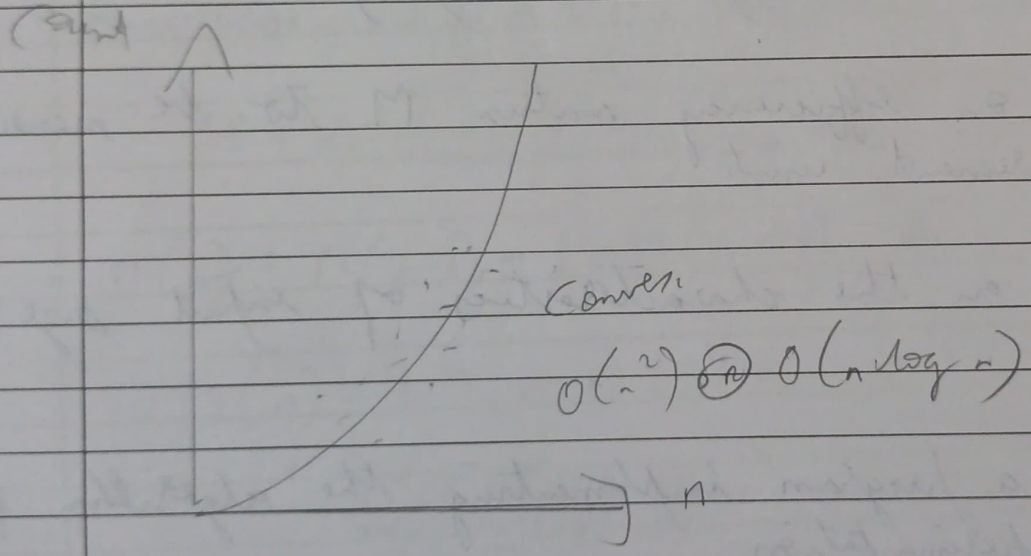
7) Analyse the data obtained

We need to plot a graph wherein x-axis

denotes the input size (n)
y-axis denotes count



Scatter Plot



(3) Master's Theorem to Solve Recurrence Relation

↓

Used to find the time complexity of recursive algorithm

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where $a \geq 1$ and $b > 1$,

Then the time complexity will be

1) If $f(n) = O\left(n^{(\log_b a - \epsilon)}\right)$ for some constant $\epsilon > 0$,
then $T(n) = \Theta\left(n^{\log_b a}\right)$

2) If $f(n) = \Theta\left(n^{\log_b a}\right)$, then
$T(n) = \Theta\left(n^{\log_b a} \log n\right)$

3) If $f(n) = \Omega\left(n^{(\log_b a + \epsilon)}\right)$ for some constant $\epsilon > 0$ & if
$a f\left(\frac{n}{b}\right) \leq c f(n)$ for some constant c

$c < 1$ & all sufficiently large n, then
$T(n) = \Theta(f(n))$

1) $T(n) = 9T\left(\frac{n}{3}\right) + n$

$f(n) = n$

$a = 9, b = 3$

$n^{\log_b a} = n^{\log_3 9} = n^2$

$f(n) = O\left(n^{2-\theta}\right)$ $\theta = 1$.

$\Rightarrow f(n) = O\left(n^{2-1}\right) = O(n)$

$T = \Theta(n^2)$

2) $T(n) = T\left(\frac{2n}{3}\right) + 1$

$f(n) = 1$

$a = 1$

$b = \frac{3}{2}$

$n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$

~~$T = \Theta(1 \log n)$~~

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

3) $T(n) = 3T\left(\frac{n}{4}\right) + n \log n$

$f(n) = n \log n$

$n^{\log_b a} = n^{\log_4 3} = n^{0.8}$

$n \log n = \Omega\left(n^{0.8+0.2}\right)$

$a f\left(\frac{n}{b}\right) \leq c f(n)$

$3 \cdot \frac{n}{4} \log \frac{n}{4} \leq c\, n \log n \leq c\, f(n)$

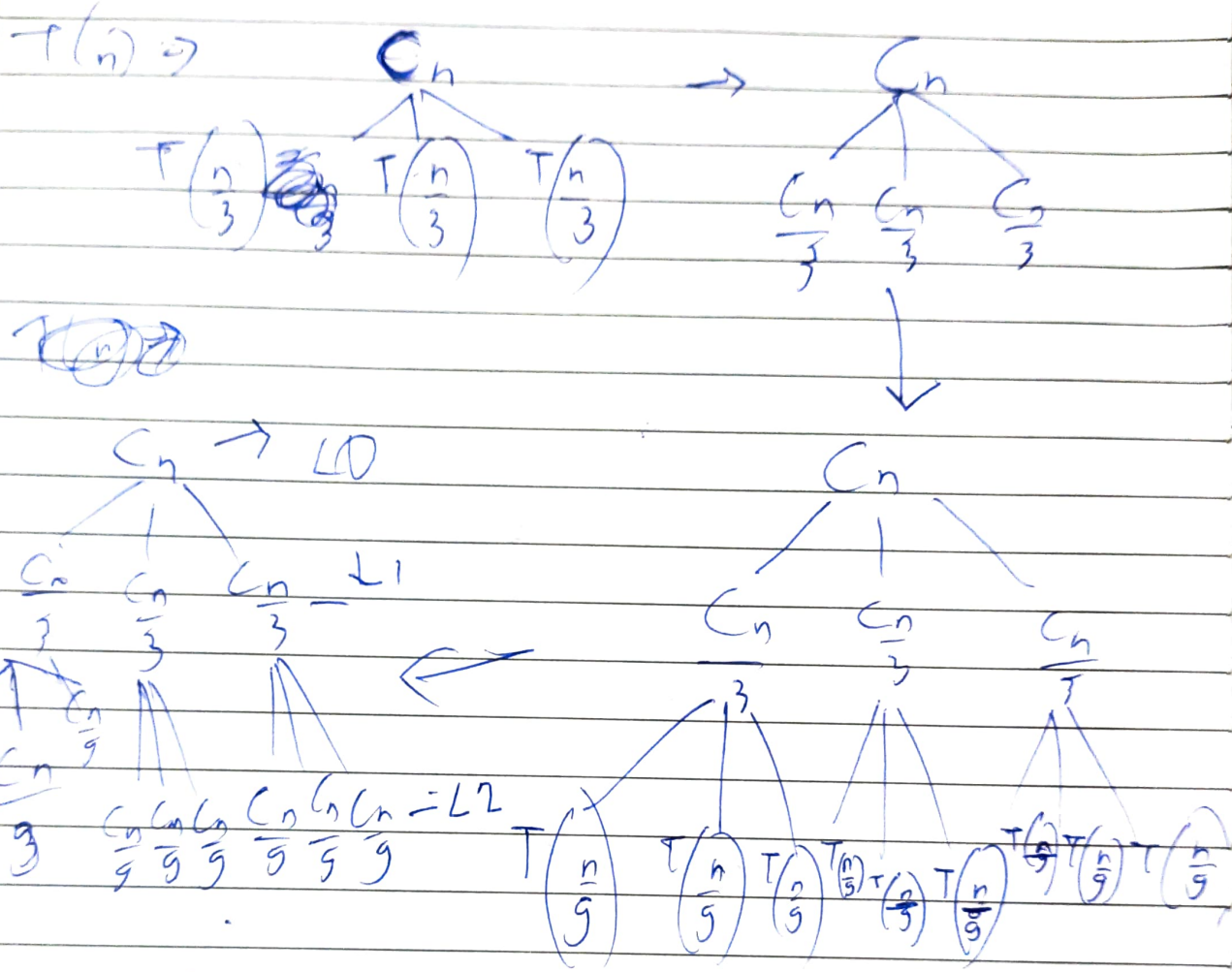
$c = \frac{3}{4}$

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

$= \Theta(n \log n)$

(4) Recursive Tree method to Solve Recurrence

$$T(n) = 3T\left(\frac{n}{3}\right) + c_n$$



No. of nodes	0	1	2	3	i
Level	3^0	3^1	3^2	3^3	3^i
Cost	c_n	c_n	c_n	$c_n \cdot 3^i T(1)$	

$\frac{n}{3^h} = 1$

$n = 3^h$

$\Rightarrow h = \log_3 n$

Total cost $T(n) = 3^h T(1) + \sum_{i=0}^{h-1} cn$

$= 3^h [T(1)] + cnh$

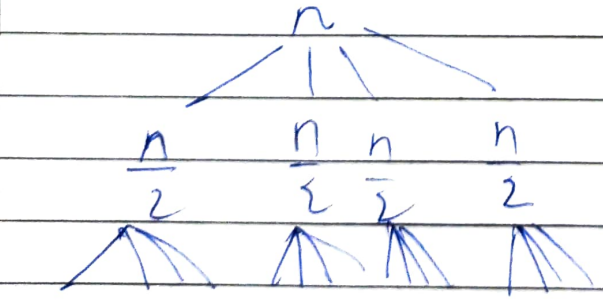
$= 3^{\log_3 n} T(1) + cn \log_3 n$

$= n^{\log_3 3} T(1) + cn \log_3 n$

$= n T(1) + cn \log_3 n \rightarrow$ dominating term

$\therefore T(n) = \theta(n \log_3 n)$

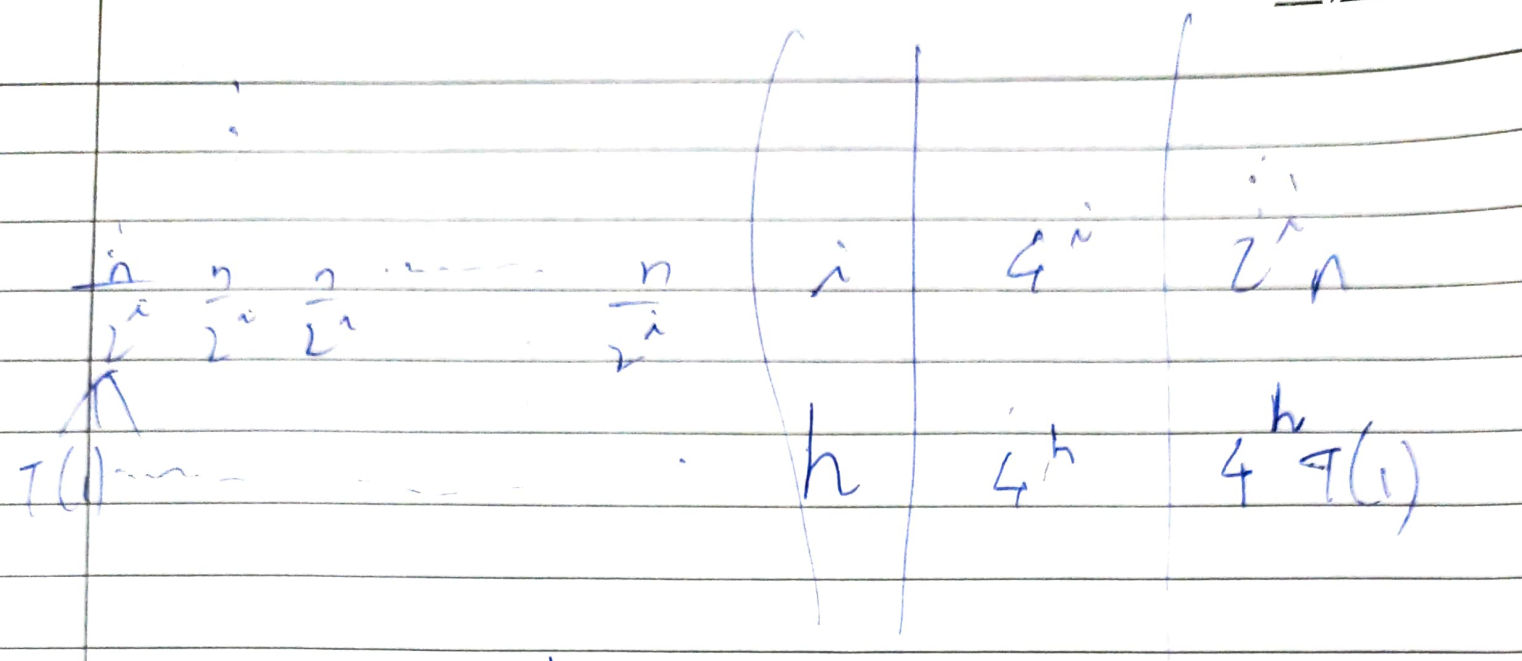
$T(n) = 4T\left(\frac{n}{2}\right) + n$



Level	No. of nodes - i	Cost
0		n
1	4^1	$2n$
2	4^2	$2^2 n$
3	4^3	$2^3 n$

$2^n - 1$

11



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

$$n = 2^h \Rightarrow h = \log_2 n$$

$$T(n) = 4^h T(1) + \sum_{i=0}^{h-1} 2^i n$$

$$= 4^{\log_2 n} T(1) + n \sum_{i=0}^{h-1} 2^i$$

$$= n^{\log_2 4} T(1) + n(1 + 2 + 2^2 \cdots 2^{h-1})$$

$$\leq n^2 T(1) + n(2^h - 1)$$

$$\leq n^2 T(1) = n^2$$

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