JAN 2024	S	M	T	W	T	F	S
		1	2	3	4	5	6
	7	8	9	10	11	12	13
	14	15	16	17	18	19	20
	21	22	23	24	25	26	27
	28	29	30	31			

DSA

1/2/24
JANUARY
'24
01st Week
001-365
01
MONDAY

→ DSA — Data Structures & Algorithms

→ Data structures — How data is store & organize in the computer. and can be used efficiently.

→ Algorithms — Sequence of finite steps to solve a particular problem.

→ Why is DSA Important?

(i) make you a better software developer. [Who is solving Prob
(ii) Helps you in getting a job. ☺
(iii) winning the sport of competitive coding. ☺

→ Roadmap to learn DSA —

(i) Learn a Programming language.
  (*) C++ (*) Java (*) Python (*) JavaScript
(ii) Learn DSA Basics and implement it.
(iii) Learn language libraries that have DSA implemented for you.
(iv) DO practice and learning together.

→ Depending upon organizing of Data, Data structure are classified into two types.

DS

Linear Data Structure
↳ Elements are stored in a Sequential order.
eg linked list, Stack, Queue
array

Non-linear Data Structure
↳ Elements are stored in non linear order.
eg Tree & graph.
Heap.

JANUARY
02
TUESDAY
'24
01st Week
002-364

Basics

S	M	T	W	T	F	S
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

JAN 2024

Analysis of Algorithms.

Algorithms = step by step process to solve a problem is called algo.

Flow chart = pictorial representation of an algorithms.

Advantage of algorithms :—

(a) problem statement will be simplified.
(b) Easy to understand the problem statement.
(c) Easy to provide implementation by using any PL.
(d) we will get a format / templat / Pattern to solve the Problem

Properties :—

(a) Every algorithms should take either zero (or) more input.
(b) Every Program should produce atleast one output.
(c) deterministic (same output should be generated even if you run again & again).
(d) Clearly written.
(e) terminate at finite steps.
(f) Efficient.

Approaches to solve an algorithms :—

→ Theoritical knowledge of the problem is essential but it is not sufficient to perform implementation & testing.

(1) Constraints / condition :—

→ Given a problem statement, constraints are very very important: first we have to identify all the constraints related to given problem

(2) Idea Generation is very very important.
→ More if you practice, you will get idea.
→ By practicing you will get a pattern of the problems.
→ Easily, we can solve unseen problem.

(3) Complexities Analysis:—
→ Finding the solutions for a problem is not sufficient.
→ we have to identify the solution which takes less time & less memory.
→ By doing analysis w.r.t time & space complexity.

(4) Coding :—
→ If you have all the above things, then we can select any PL & solve the given by using proper syntax & semantics.

(5) Testing :—
After completion of Program, validation can be done by applying various inputs to the implemented program.

Sample Algorithms & Implementations :—

Input Size	Time Complexity
$n > 10^8$	$O(\log n)$, $O(1)$
$n \leq 10^8$	$O(n)$
$n \leq 10^6$	$O(n \log n)$
$n \leq 10^4$	$O(n^2)$
$n \leq 500$	$O(n^3)$
$n \leq 25$	$O(2^n)$
$n \leq 12$	$O(n!)$

Increasing order of TC

1s ≈ 10^8 operation.

→ Time Complexity doesn't have units like minutes and second. Instead, it tells you how the running time grows with input size, regardless of actual time.

S	M	T	W	T	F	S
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

JAN 2024

Analysis of Recursion

```
def fun(n):                    # n
    if n==1:                   # T(1) = Θ(1)
        return                 # T(n) = 2T(n/2) + Θ(n)
    for i in range(n):         #      = Θ(n)
        print('GFG')
    fun(n/2)                   # n/2
    fun(n/2)                   # n/2
```

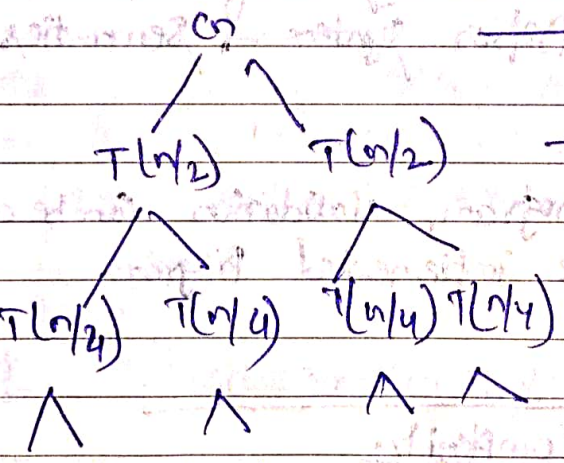
Q9

Recursion Tree method for finding T.C

$T(n) = 2T(n/2) + cn$

$T(1) = c$

Sol^n



Cost/work done

cn — cn

cn — cn

cn — cn

Just compute the total work done

$cn + cn + cn + \cdots + cn$ ($\log_2 n$ terms)

$= \Theta(n \log_2 n)$ ✓

(*) We work on Recursion Tree Method using 2 steps.

(1) We write Non recursive part as root of tree and recursive parts as childrens.

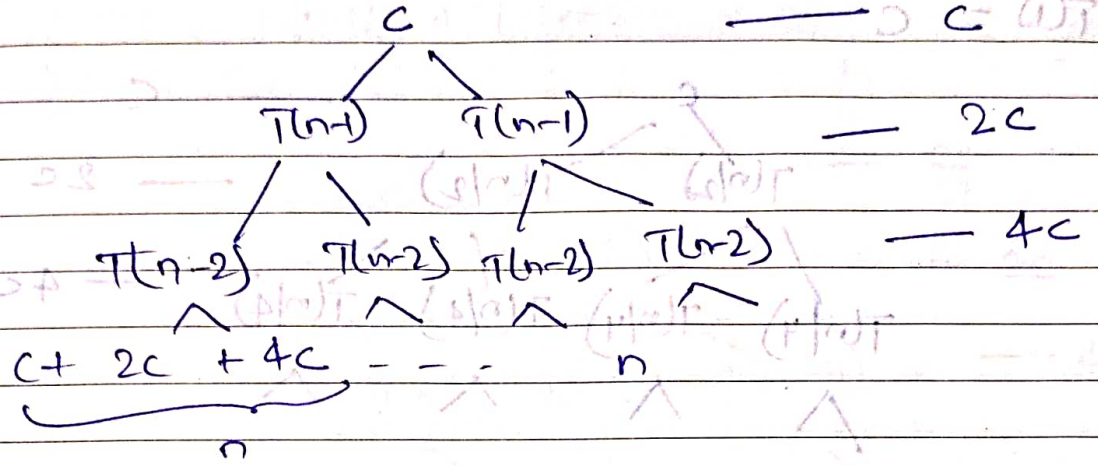
(2) We keep expanding children until we see a pattern.

(3) Just add height of the tree and that would be your T.C.

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

$T(1) = c$

Sol^n



$c + 2c + 4c - - - n$

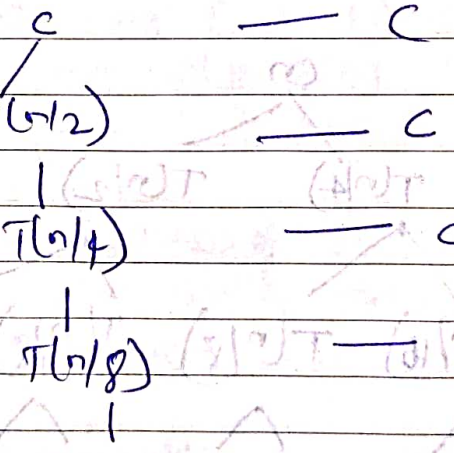
n

$$\frac{a(r^n - 1)}{r - 1} = 1 \times (2^n) = \Theta(2^n)$$ ✓

eg $T(n) = T(n/2) + c$

$T(1) = c$

Sol^n



$c + c + c + c$

$\log n$

$= \Theta(\log_2 n)$ ✓

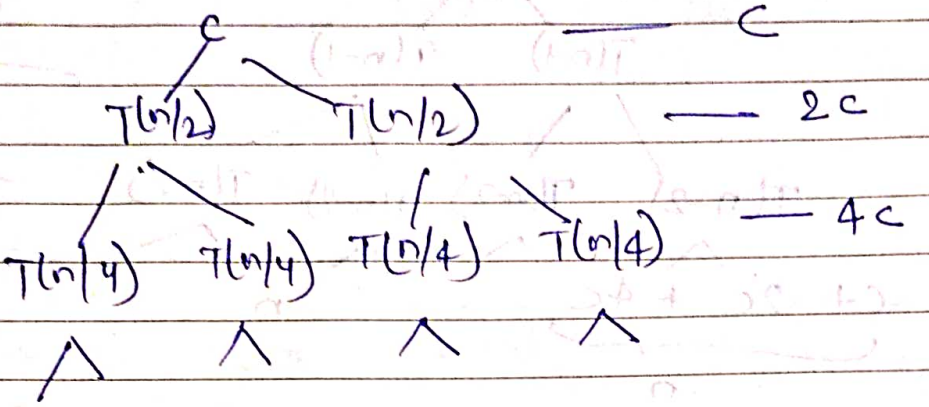
S	M	T	W	T	F	S
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

JAN 2024

eg $T(n) = 2T(n/2) + c$

$T(1) = c$

Sol°

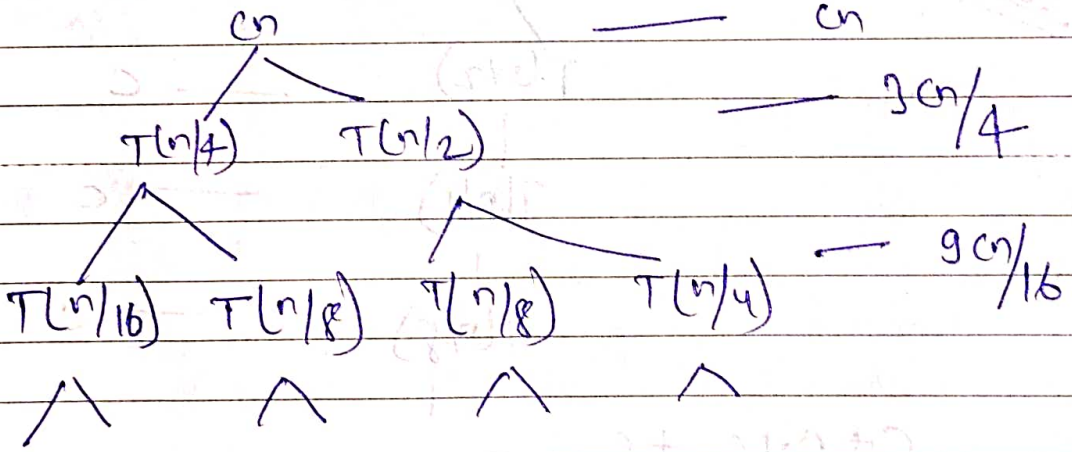


$c + 2c + 4c + \cdots$ ($\log_2 n$ terms) $= \frac{c(2^{\log_2 n} - 1)}{2-1} = O(n)$ ✓

eg $T(n) = T(n/4) + T(n/2) + cn$

$T(1) = c$

Sol°



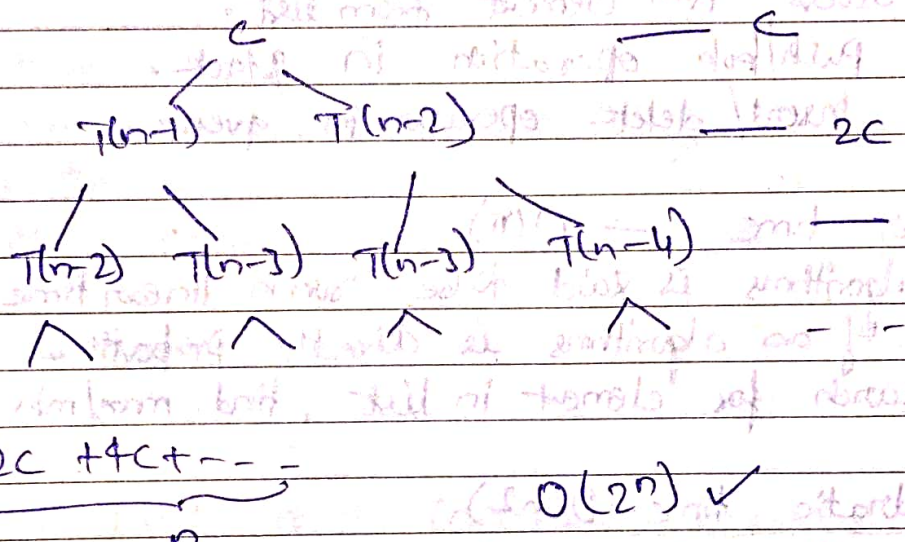
$cn + \frac{3cn}{4} + \frac{9cn}{16} + \cdots$ ($\log_{\frac{4}{3}} n$ terms)

Or $O\left(cn \times \frac{1}{1-3/4}\right) = O(n)$ ✓

eg ~~T(n) = T(n/4) + T~~

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

$T(1) = c$



$c + 2c + 4c + \cdots$ (n terms)

$O(2^n)$ ✓

Asymptotic Notation / Asymptotic analysis :-

1. Big-oh Notation → O (≤) [upper Bound]
2. Omega-Notation → ω/Ω (≥) [lower Bound]
3. Theta-Notation → Θ (=) [Tight Bound]

1. Worst case : 'O', max No. of steps required, (upper bound)
2. Best case : 'Ω', min No. of steps required, (lower ")
3. Average case : 'Θ', Average " " " , (tight ")

lower Bound | ≤ Average | tight ≤ upper bound

$\Omega < \Theta < O$

always Prefered 'O' bound because it will cover all the Edge test cases.

JANUARY
08
'24
MONDAY
02nd Week
008-358

S	M	T	W	T	F	S
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

Growth of Functions

(1) Constant time – O(1)

→ An algorithm is said to be run in constant time if the output is produced in constant time, irrespective of input size.

eg access nth element from list. [indexing concept]
Push/pop operation in stack. [direct append]
Insert/delete operation in queue. [direct append]

(2) Linear time – O(n)

→ An algorithms is said to be run in linear time if the execution time of an algorithms is directly proportional to input size.

eg search for element in list, find max/min in list.

(3) Quadratic time – O(n2)

→ An algorithm is said to be run in quadratic time if it is executed time of an algo is directly proportional to input size.

eg selection sort, insertion sort, bubble sort etc.

(4) Logarithms time – O(logn)

→ An algo is said to run in logarithm time if the execution time of an algo is directly proportional to the logarithms value of input size.

eg divide & conqure, binary search.

(5) N-logarithms time – O(nlogn)

→ An algo is said to run in n × logarithms time if the execution time of an algo is directly proportional to the logarithm is value of input of input size multiplied with input size.

→ Time taken to execute No. of instruction per second is called TC.

O(n) ↳ No. of element & size of element

FEB 2024

S	M	T	W	T	F	S
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29		

JANUARY '24

02nd Week 009-357

09 TUESDAY

(6) Exponential time — $O(n^2)$

→ All possible subsets / sub combination of element of input data

eg Power set, Subset.

09

(7) Factorial time — $O(n!)$

10 In this algorithms, all possible permutation of elements of input data are generated.

11 eg abc = abc, acbi bac, bca, cab, cba, → $6 = 3! = 3!$

12 $C < \log\log n < \log n < n^{1/3} < n^{1/2} < n < n^2 < 2^n < n^n < n!$

01

Direct way to Find and Compare Growths.

02 (1) Ignore lower order terms.

(2) Ignore leading term Constant.

03

04 # Space complexity :— [How much memory particular program is taking]

Order of growth of memory (RAM) usage in terms of input.

05

eg def get sum(n):
06 return n*n/2

07 → SC = O(1)

eg def getsum(n):
sum = 0
i = 1
While i <= n:
sum = sum + i
i = i + 1
return sum

SC = O(1)

→ Auxiliary Space —

→ Order of growth of extra space (space other than input/output).