ALGORITHMS (12 mrks) (C + DS + Algo ⇒ approx 20-25 mrks)

Syllabus:-
1) Finding Time complexity (*V.Imp)
   Finding space complexity
   1-2 mrk
2) Asymptotic Notation (*V.Imp)
   → O, Ω, Θ
   approx 4 mrk
3) Divide and Conquer (DAC) 4mrk
4) Greedy Technique 4mrk
5) Dynamic Programming 4mrk.
   (Imp)

Algorithm

Def^n:- It is a combination of sequence of finite steps to solve a problem.

Eg:-
```
mul()
{
  ① Take 2 numbers (a, b)
  ② Multiply a*b and store in c
  ③ return(c)
}
```

* Properties of algorithm.

(1) It should terminate after finite time
   - Finite steps in any algorithm does not guarantee finite time of running that algo.
   - But every algo. should contain finite steps.

(2) It should produce atleast 1 output
(3) It should be deterministic.
- Any algo. should not give multiple answers for a particular input / It should not be ambiguous.
(4) It is independent of Programming languages.
(5) Every statement should be effective.

* Designing Algorithm. (Steps needed to design algo.)

(1) Problem Defination (SRS document)
(2) Select design technique →
- ~~Divide & Conquer~~
- Greedy Method
- Dynamic Prog.
- Backtracking etc

* Note :- Algorithm design for any problem is nothing but selecting appropriate design technique (template) to solve that problem.

(3) Draw Flow Chart
(4) Testing - (check that for a input, correct output is there or not) (Hell many)
(5) Implementation (coding)
(6) Analysis (analyze time & space complexity)

Design and Analysis of Algorithm - Imp areas of CSE - only CSE people can do.

* Purpose of studying Algorithms :- Not to write entire program from scratch but to correctly determine which technique / template will give accurate results in less time & space.

Analysis
(1) Time Complexity [CPU - time]
(2) Space complexity [main memory]

† when a problem has multiple sol^ then only we will analyze and choose a better sol^

(1) * Time Complexity (based on 2 things)

(2 marks every year)

Time Complexity (Program) = Compile time (Prog.) + Running time (Prog.)

based on → Compiler → S/w → Language of compiler matters (eg:- C is faster than Java)

based on → Processor → H/w → Type of H/w matters (I9 processor is faster than I_3)

Analysis

Postiary Analysis
→ Postpone the answer i.e ask more questions before giving the answer

Prion Analysis
→ Give direct answer without postponing / asking more questions.

Postiary Analysis (supercomputer matters)	Apriori Analysis (super logics matter)
① It is dependent on lang. of compiler & type of Hardware	① It is independent of lang. of compiler & type of Hardware.
② More Accurate	② Approximate
③ System to system answer will change (based on systems h/w & s/w)	③ same answer given - independent of the system
→ If a program running faster then reason is that, system has good h/w & s/w	→ If a program running faster, reason is that developer has good logics
∴ No industry follows this analysis	∴ It is followed in industry (developers make a company big) not h/w or s/w

Start of Big-O

* Apriori Analysis
It is a determination of order of magnitude of statement.

Eg ①:-
```
main()
{
1. x = y+z;
}
```
⇓ O(1)

whenever this program is running, this statement will run how many times? Ans = 1 ⇒ this is called order of magnitude of a statement

Eg ②:-
```
main()
{
1. x = y+z ⇒ 1
2. a = b+c ⇒ 1
3. c = A+e ⇒ 1
}
```
1+1+1 = 3 ⇒ every constant can be written as O(1)

⇓ 3 = O(1)

Eg ③:-
```
main()
{
①- x = y+z;
   for (i=1; i<=n; i++)
②- x = y+z;
③- a = b+c;
}
```
⇒ In this program, for loop contains no brackets, means only 1st line after loop i.e. x=y+z (statement ②) belongs to for loop.

⇒ This program contains 4 lines of code but only 3 statements.

1st statement runs 1 time
2nd stat. runs → n times } order of magnitude of each statement
3rd stat. runs 1 time

n+2 = O(n)

Note :- 'n' is always a very big no. until and unless it is mentioned in the program.

④:- main()
{
① — x = y+z;
for (i=1; i≤n; i++)
② — x = y+z;
for (i=1; i≤n; i++) outer loop
for (j=1; j≤n; j++) inner loop
③ — x = y+z;
}

running → 1 times
running → n times
⇒ This code contains 3 statements

i=1	i=2	... i=n
j=1,2,3...n	j=1,2...n	j=1,2...n
n+	n+	... n times

= n × n
= n²

1 + n + n² = O(n²)
actual ans ⇓ approx ans

Note :- Time complexity means working on loops. If many loops are there, go for larger loop. If no loop, it means constant time

... above program, loops are those which needs to run multiple ...es ; ∴ we keep these loops in cache memory.
... that pt. of time whatever is imp. for the program, that lines ...ode will be in cache memory (locality of reference)

⑤: main()
{
while (n>1) ⇒ The no. of times this loop will run depends on value of n
{
z = y+z;
y = a+b; ∴ This loop runs n times
n = n-1; ∴ order of magnitude = n
}
}
⇒ O(n) ⇒ Time complexity.

Eg:- main()
{
while (n>1) ⇒ since n = n-2, this while loop runs n/2 times (if it was n-100 ⇒ it runs n/100 times)
{
x = y+2;
n = n-2
}
}
∴ Time complexity ⇒ n/2 = O(n)

Eg:- main()
n=100
{
while (n>1) ⇒ In this prog. value of n is not decreasing
{
x = y+2; ∴ This prog. runs infinitely
}
} ∴ This is first of all, not a Algorithm
∴ no Time complexity (algo. should terminate)

Eg:- main()
{
while (n>1) } loop runs n/(10+90) = n/100 times = 1/100 · n
{
x = y+2;
n = n-10;
n = n-90; } Time complexity = O(n)
}

Eg:- main()
{
i=1
while (i≤n) ⇒ } loop runs n times
{
x = y+2; } n/1 = O(n)
i = i+1;
}
}

Eg:- main
{
i=12
while (i≤n)
{
x = y+2; } n/50 - 12
i = i+50;
}
} O(n)
focus on loop only

Note :- Here initially i=1 (small value) ∴ we did i=i+1
In prev. ques n = very big value ∴ we did n=n-1
↳ If we do reverse then prog will go into a loop and it will not be an Algorithm.

Eg:- main()
{ i=0
while(i<n)
{ x=y+2;
i=i+1;
i=i+4;
i=i+190;
}
i=i+200
}
this loop runs $\frac{n}{200}$ times
$= O(n)$

Eg:- main()
{ i=1
while(i<n)
{ x=a+b;
i=i+7;
i=i+83;
i=i-50;
}
}
$\frac{n}{40} = O(n)$
overall i=i+40
i value increases
∴ it will not go into inf. loop

Eg:- main()
{ i=1
while(i<n)
{ x=y+2;
i=10*i;
}
}
$\Rightarrow 10^k = n$
$k = \log_{10}(n)$
T.C = $O(\log_{10} n)$

Eg:- main()
{ i=1
while(i<n)
{ i=10*i;
i=5*i;
}
}
i=50*i
∴ Time Complexity = $O(\log_{50} n)$

Eg:- main()
{ i=1
while(i<n)
{ a=b+c;
i=i*2;
}
}
To solve this problem, assume n=64
i=1 ⇒ ∴ 1 ≤ 64 ✓
2 ≤ 64 ✓
4 ≤ 64 ✓
8 ≤ 64 ✓
16 ≤ 64 ✓
32 ≤ 64 ✓
64 ≤ 64 ✓
128 ≤ 64 ✗
loop runs total 7 times
i.e $\log_2(64)+1$ times
∴ Time Complexity = $O(\log_2 n)$

Proof:- we run the loop. and as i value increases, it comes close to n. And when it is > n we stop the look. let loop runs k times till i = n.

i value gets multiplied by 2 with every loop iteration ⇒ $1 = 2^0$, 2, 2^2, 2^3, ... (k times), $2^k = n$

⇒ $\log_2(2^k) = \log_2(n)$
$k = \log_2 n$

Eg:- main()
{ i=0
while(i<n)
{ i=2*i;
}
}
Here i=0
∴ This will go into ∞ loop
∴ Not a algorithm

Eg:- main()
{ i=17
while(i<n)
{ x=a+b;
i=2*i;
i=25*i;
}
}
i=150n
i = 17
17×150
17×(150)^1
17×(150)^2
k times
17×(150)^k
$17 \times (150)^k = n$
$\log_{150}(150)^k = \log_{150}(\frac{n}{17})$
$k = \log_{150} n - \log_{150}(17)$
∴ Time comp. = $O(\log_{150} n)$

Eg:- main()
{ while(n>1)
{ n=n/2;
a=b+c;
}
}
⇒ Here, n value keep on decreasing till it reaches n=1
∴ n value = n, $n/2$, $n/2^2$, ... loop k times ..., $n/2^k$
⇒ Time comp ⇒ $O(\log_2 n)$

$\frac{n}{2^k} = 1 \Rightarrow n = 2^k$
$\log_2(n) = \log_2(2)^k$
$k = \log_2(n)$

Eg:- main()
{ while(n>79)
{ n=n/5
n=n/7
n=n/2
} n=n/70
∴ Time comp = $O(\log_{70}(n))$
}

Eg:- main()
{ while(n>79)
{ n=n/5
n=n/7
n=n/2
} n=n/70 ; n=n/70 × 5 = n/14
n=n×5
}
∴ T.C = $O(\log_{14} n)$

Eg:- main()
{ while(n>79)
{ n=n/5
n=n-5
n=n/7
n=n+6
n=n×5
n=n/2
n=n×7
} n=n/2 ⇓ $O(\log_2 n)$
Here ignore +, − as they will make a very small change compared to ×, ÷

Eg:- main()
{ i=71;
while(i<n)
{ i=i^34;
x=a+b;
}
}
71 ⇒ 71^{34^0}
$(71)^{34}$ ⇒ 71^{34^1}
$((71)^{34})^{34}$ ⇒ 71^{34^2}
71^{34^3}
K times.
$71^{34^K} = n$
∴ Time comp = $O\left[\log_{34}(\log_{71} n)\right]$

Eg: main()
{ i=2;
while(i<n)
{ i=i^2;
z=y+z;
}
}
$2 = 2^{2^0}$
$(2)^2 = 2^{2^1}$
$(4)^2 = 2^{2^2}$
$(16)^2 = 2^{2^3} = 2^{2^3}$
k times
$2^{2^k} = n$
$\log_2(2^{2^k}) = \log_2(n)$
$\log(2^k) = \log_2(\log_2 n)$
$k = \log_2(\log_2 n)$
∴ T.C = $O\left[\log_2(\log_2 n)\right]$

Eg: main()
{ i=67
while(i<n)
{ i=i^2 ; i=(i^2)^3
i=i^3 = i^6
y=i+2
}
}
ignore small small things
$O\left[\log_6(\log_{67} n)\right]$

Eg:- main()
{ while(n>2) ⇒ n
{ n=n^1/2 ; $n^{1/2}$
a=b; $n^{1/2^2}$
} $n^{1/2^3}$
$n^{1/2^k} = 2$
⇒ $\frac{1}{2^k}\log_2 n = \log_2 2$
$\log_2 n = 2^k$ ⇒ $k = \log_2(\log_2 n)$
Time comp = $O\left[\log_2(\log_2 n)\right]$

Eg: main()
{ while(n>42)
{ n=n^1/2
n=n^1/5
n=n^1/7
n=n×2
} $n = n^{1/2 \times 1/5 \times 1/7 \times 2}$
⇒ $n = n^{1/85}$
⇓
$O\left[\log_{35}(\log_{43} n)\right]$
n=n-2
n=n+4
ignore small small things

Eg:- main()
{ for(i=1; i≤n; i=i+7)
{ x=y+z;
a=b+c;
}
}
flow of code
$i = 1$
$\Rightarrow i = 1 + 7$
$= 1 + 2 \times 7$
$= 1 + 3 \times 7$
$= 1 + k \times 7 = n$
$k = \frac{n-1}{7}$ ⇒ $O(n)$

GATE Que) main()
{ p=0, q=0
for(i=1; i≤n; i=2×i) → This loop runs $\log_2 n$ times
p++; ∴ p increases as much as loop runs
∴ $p = \log_2 n$
for(j=1; j≤p; j=2×j) → This loop runs $\log_2 p$ times
q++;
}
what is q value? & time comp:?
$\because q = \log_2 p = \log_2(\log_2 n)$
$q \overset{\text{approx}}{=} O\left[\log_2(\log_2 n)\right]$

ow, for time complexity we consider that loop which takes time
1st loop : $\log n$, 2nd loop $= \log(\log n)$ run times

bigger $\Rightarrow$ Time comp $= O[\log n]$

e) Find Time complexity.

```
main ()
{
  for(i=n ; i>10 ; i=i^(1/4))          ⇒ log_4(log_10 n)
  {
    for(j=201 ; j<=n^5 ; j=j+400)      ⇒ n^5/400
    {
      for(k=47 ; k<=n^84 ; k=k*log5)
      {
        1<k<k^61  ⇒ log_61(log_47 n^84)
      }
    }
  }
}
```

ignore small-small things as compared to k^{61}

$\therefore$ Time Comp. $= O\left[\log_{61}\log_{47} n^{84} \times \frac{n^5}{400} \times \log_4 \log_{10} n\right]$

* Note:- In this Question, for Time comp. we took multiplication of all loops because there were inner loops for each 'for loop'.

But in prev. que, we took Time complexity which was bigger as our final Answer because both were outer loops and were not directly dependent.

$\therefore$ Time comp. of biggest loop was answer. But here for dependent-loops, multiplicat^n is answer.

Que) Find Time complexity. and value of x = ?

```
main ()
{
  x=0
  for(i=79 ; i<=n^25 ; i=i^3)
  {
    j=1
    while(j<=n)
    {
      j=2*j;
      x=x+7;
    }
  }
}
```

Sol^n Time complexity $= O\left[\log_3 \log_{79} n^{25} \times \log_2 n\right]$ ('for loop' 'while loop')

x-value $\Rightarrow$ when 'i' value = 79, 'j' loop runs $\log_2 n$ times and value of x becomes $7+7+7+\cdots+7 = 7\times\log_2 n$

Next 'i' value becomes i^3 ie $(79)^3$, 'j' loop runs again $\log_2 n$ times again x value increases $\Rightarrow$ $\underbrace{7\log_2 n}_{i=79} + \underbrace{7\log_2 n}_{i=(79)^3}$

and so on i loop runs $\log(\log n)$ times

$\therefore$ x value $= 7\log_2 n + 7\log_2 n + \cdots\cdots 7\log_2 n$

$= 7\log_2 n \times \log_3 \log_{79} n^{25} = O\left[7\log_2 n \times \log_3 \log_{79} n^{25}\right]$

Que) main ()

```
{
  for(i=1; i<=n; i++) → loop runs n times
  {
    for(j=1; j<=i; j++) → loop runs i times (dependent on previous loop)
    {
      x=y+2;
    }
  }
}
```

inner loop

for each i value	i=1	i=2	i=3 ---- i=n
loop ⇒ j=	1	1,2	1,2,3 ---- 1,2---n

this stmt runs $\therefore T.C = 1+2+3+\cdots$ n times this statement runs

$= n(n+1)/2$

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

Que) main()

```
{
  for(i=1; i<=n; i++)          → loop runs n/1 times
  {
    for(j=1; j<=n; j=j+i)      → loop runs n/i times
    {
      x = y+2;
    }
  }
}
```

i=1	i=2	i=3	----	i=n
j=n/1	j=n/2	j=n/3		j=n/n

∴ Time comp. = $n + \frac{n}{2} + \frac{n}{3} + \frac{n}{4} - - - \frac{n}{n}$

$= n\left[\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} - - - + \frac{1}{n}\right] = n \log n$

∴ T.C = $O[n \log n]$

Que) main()

```
{
  for(i=1; i<=n; i++)
  {
    if(n%2==0)
    {
      for(j=1; j<=n; j++)
        x = y+2;
    }
  }
}
```

Here if n = even no

⇒ i=1 ⇒ 'j' loop runs n times

i=2 ⇒ 'j' loop runs n times

i=3 ⇒ — 11 —

i=n ⇒ — 11 —

∴ Time comp = $O(n^2)$

If n = odd ⇒ i=1 → 'j' loop does not run

i=2 → — 11 —

i=n → — 11 —

∴ Time comp = $O(n)$

There are 2 Time comp.

∴ worst case = $O(n^2)$, if n is even.

Best case = $O(n)$, if n is odd.

1-2 mins / Prev Years

(2) Asymptotic Notation

Let f(n) and g(n) be 2 positive functions.

* Note that here n is +ve integer number greater that 0.

1. Big-Oh (O)
2. Omega (Ω)
3. Theta (Θ)

If we say that betⁿ 2 fⁿˢ theta (Θ) is possible. Then we can conclude that betⁿ those 2 fⁿˢ Big-Oh(O) (greater than, equal to) and Omega (Ω) possible (less than, equal to) (equal to)

* Big-Oh :- we can say $f(n) = O(g(n))$ iff and only if $f(n) \le c \cdot g(n)$

eg:- $n^2+n+1 = O(n^2) = O(n^3) = O(n^{10})$

$n^2 \le n^2$, $n^2 \le n^3$, $n^2 \le n^{10}$ ⇒ $a = O(b)$ ⇒ $a \le c \cdot b$ (c = constant)

∴ $f(n) = O(g(n))$ iff $f(n) \le c \cdot g(n)$, ∀n where $n > n_0$ such that ∃ two +ve constants $c > 0$ and $n_0 \ge 1$

Basically, Big-Oh means ⇒ Prove right hand side bigger, by taking constant (c) help of a constant (c)

Que)

* Omega (Ω) : we can say $f(n) = \Omega(g(n))$ iff and only if $f(n) \geq c \cdot g(n)$ $\forall n$ where $n \geq n_0$ such that there two constants $c > 0$ and $n_0 \geq 1$

• Big-oh Questions

Que 1) $f(n) = n^2+n+1$, $g(n) = n^2$ prove that $f(n) = O(g(n))$ ⇒ to satisfy this cond^n

let $n^2+n+1 \leq c \cdot (n^2)$, $\forall n$, $n \geq n_0$

let us take max value for each n^2+n+1

$n^2+n^2+n^2 = 3n^2$

$\therefore n^2+n+1 \leq 3n^2$ this cond^n satisfies for all n values ≥ 1

Hence, we can say that

$\boxed{n^2+n+1 = O(n^2)}$ $(c, n_0) = (3, 1)$

originally left side is bigger but we proved right hand side bigger by taking help of a constant.

Que 2) $f(n) = n+10$, $g(n) = n-10$ prove that $f(n) = O(g(n))$

To satisfy this cond^n let ~~f(n)~~ $n+10 \leq c \cdot (n-10)$

if we are able to prove $n+10$ always less than $c \cdot (n-10)$ for some value of constant c, then we can say that $f(n) = O(g(n))$

By observ^n we can say that $n+10$ is bigger than $n-10$ but with help of a constant c, we will prove $n+10 < c \cdot (n-10)$

So, let $1 \cdot n+10 \leq \underset{\Downarrow 2}{c} (n-10)$, $\forall n$, $n \geq n_0$

The above rel^n satisfies for all values of $n \geq 30$ (n_0)

$\therefore (c, n_0) = (2, 30)$ i.e $n = 30, n = 31, n = 32 \ldots$

let $1 \cdot n+10 = \underset{\Downarrow 3}{c} (n-10)$, $\forall n$, $n \geq n_0$

The above rel^n satisfies for all values of $n \geq 20$ (n_0)

$\therefore (c, n_0) = (3, 20)$

Hence, by above we proved that there exist a (c, n_0) for which $n+10 \leq c \cdot (n-10)$ ∴ we can conclude that

$\boxed{n+10 = O(n-10)}$

$\Updownarrow$

$n+10 \leq c \cdot (n-10)$ $\forall n$, $n \geq n_0$ if $c = 20$, $n_0 \geq 90$

Que 3) $f(n) = n^2$ $g(n) = n^2+n+1$ P.T $f(n) = O(g(n))$

Sol^n

let $n^2 \leq \underset{\Downarrow 1}{c} \cdot (n^2+n+1)$ $\forall n$ $n \geq \underset{\Downarrow 1}{n_0}$

Hence we can say that $\boxed{n^2 = O(n^2+n+1)}$ $(c, n_0) = (1, 1)$

Que 4) $f(n) = n^2$ $g(n) = n$ P.T $f(n) = O(g(n))$

~~Let~~ To prove $f(n) \overset{\Updownarrow}{=} O(g(n))$

let $n^2 \leq c \cdot (n)$, $\forall n$, $n \geq n_0$

$= n \times n \leq c \cdot n$

⇒ Not possible any value of c

⇒ Here we cannot take $c = n$ as c is constant and not a function

$\Rightarrow \boxed{n^2 \neq O(n)}$

• omega Questions

Que 1) $f(n) = n^2+n+1$, $g(n) = n^2$ P.T $f(n) = \Omega\, g(n)$

Soln To prove $f(n) = \Omega\, g(n)$

Let $n^2+n+1 \geq c \cdot (n^2)$ $\forall n$, $n \geq n_0$

Here we cannot take c value greater because it will start increasing right hand side.

$\therefore$ take $c = 1$ and $n_0 \geq 1$

$\therefore n^2+n+1 \geq 1 \cdot (n^2)$ $\forall n$, $n \geq 1$

Hence $\boxed{n^2+n+1 = \Omega(n^2)}$ $(c, n_0) = (1, 1)$

Que 2) $f(n) = n+10$, $g(n) = n-10$ P.T $f(n) = \Omega\, g(n)$

Let $n+10 \geq c \cdot (n-10)$ $\forall n$, $n \geq n_0$

Here anyways left side i.e $n+10$ is bigger than $n-10$ $\therefore$ take $c = 1$

$\therefore$ The cond^n satisfies for all values of $n_0 \geq 10$

$\therefore n+10 \geq 1 \cdot (n-10)$ $\forall n$, $n \geq 10$ (since $(n-10)$ can not be -ve)

Hence $\boxed{n+10 = \Omega(n^2)}$

$(c, n_0) = (1, 10)$

Que 3) $f(n) = n^2$ $g(n) = n^2+n+1$ P.T $f(n) = \Omega\, g(n)$

Soln To prove $f(n) = \Omega\, g(n)$

Let $n^2 \geq c \cdot (n^2+n+1)$ $\forall n$, $n \geq n_0$

Hence there exist (c, n_0) for which above cond^n satisfies

$\therefore$ we can say that $\boxed{n^2 = \Omega(n^2+n+1)}$

By looking at the problem we can say right hand side bigger but by taking help of a constant c we proved left side bigger here c can be fraction also. $(c = 1/2)$

Que 4) $f(n) = n^2$, $g(n) = n$ P.T $f(n) = \Omega\, g(n)$

Soln Let $n^2 \geq cn$ $\forall n$ $n \geq n_0$

Hence $\boxed{n^2 = \Omega(n)}$

Conclusion:- Between two f^{ns} if Big-oh (O) fails, then omega (Ω) should pass definitely.

*Note

If suppose Big oh (O) fails and omega (Ω) also fails then that two f^{ns} are not comparable

Que 5) $f(n) = n$ $g(n) = n^2$

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

$n \leq c \cdot n^2$ $\forall n, n \geq n_0$

Hence $\boxed{n = O(n^2)}$

Que 5) $f(n) = n$, $g(n) = n^2$

P.T $f(n) = \Omega(g(n))$

$n \geq c \cdot n^2$

$n \geq c \cdot n \cdot n$

not possible. cannot take c as c is constant and not f^n

$\boxed{n \neq \Omega(n^2)}$

* Theta (Θ) : We can say $f(n) = \Theta(g(n))$ iff and only if

① $f(n) \leq c_1 \cdot g(n)$ and ② $f(n) \geq c_2 \cdot g(n)$ } $\forall n, n \geq n_0$

both cond^n should satisfy

such that $\exists$ three +ve constants $c_1 > 0$, $c_2 > 0$ and $n_0 \geq 1$.

Que 1) $f(n) = n^2+n+1$, $g(n) = n^2$ P.T $f(n) = \Theta(g(n))$

Sol^n ① $n^2+n+1 \leq c_1(n^2)$, $\forall n, n \geq n_0$ $\therefore n^2+n+1 = O(n^2)$

($c_1 = 3$, $n_0 = 1$)

② $n^2+n+1 \geq c_2(n^2)$, $\forall n, n \geq n_0$ $\therefore n^2+n+1 = \Omega(n^2)$

($c_2 = 1$, $n_0 = 1$)

Above two condit^n satisfies $\therefore$ $\boxed{n^2+n+1 = \Theta(n^2)}$

$c_1 = 3$, $c_2 = 1$, $n_0 = 1$

Que 2) $f(n) = n+10$ $g(n) = n-10$ P.T $f(n) = \Theta(g(n))$

Sol^n ① $n+10 \leq c_1(n-10)$ $\forall n, n \geq n_0$ $\therefore n+10 = O(n-10)$

($c_1 = 2$, $n_0 = 30$)

and

② $n+10 \geq c_2(n-10)$ $\forall n$ $n \geq n_0$ $\therefore n+10 = \Omega(n-10)$

($c_2 = 1$, $n_0 = 10$)

$\therefore$ we can say that $\boxed{n+10 = \Theta(n-10) \ \forall n, n \geq n_0, \ c_2 = 2, \ c_2 = 1, \ n_0 = 30}$

Que 3) $f(n) = n^2$ $g(n) = n$ P.T $f(n) = \Theta(g(n))$

① $n^2 \leq cn$, $\forall n, n \geq n_0$

$\Rightarrow n \times n \leq c \cdot n$

$\therefore n^2 \neq O(n)$

② $n^2 \geq c \cdot n$ $\forall n, n \geq n_0$

$\therefore \boxed{n^2 = \Omega(n)}$

Since, here Big-oh not possible, we can conclude that $f(n) \neq \Theta(g(n))$

$\Rightarrow \boxed{n^2 \neq \Theta(n)}$

*NOK :-

From above, the fns are not asymptotically equal (theta) (equal gone α) but they are strictly.

$\boxed{n^2 > n}$ $\Rightarrow$ This is called small omega.

Que 4) $f(n) = n$ $g(n) = n^2$

Sol^n ① $n \leq cn^2$, $\forall n, n \geq n_0$ ($c_1 = 1$, $n_0 = 1$) $\Rightarrow \boxed{n = O(n^2)}$.

② $n \geq c \cdot n^2$ $\forall n, n \geq n_0$

$\Rightarrow n \geq c \cdot n \times n$

$\therefore n \neq \Omega(n^2)$

Here omega not possible

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

Equal gone α.

*NOK :-

Above fns are asymptotically not equal but they are strictly $\boxed{n < n^2}$ $\Rightarrow$ This is called small-oh

** We can conclude that :-

If b/w two fns theta (Θ) not possible $\Rightarrow$ Then it is either small-oh (or) small-omega.

Suppose both of these also not there then the fns are not comparable

Remember: ① Small-oh possible ($<$) means Big-Oh also possible ($\leq$)

∴ Every small-oh is Big-oh but not vice-versa

Similarly

② Small-omega possible ($>$) means Big-omega also possible ($\geq$)

∴ every small-omega is Big-omega but not vice versa

---

* Note:

① Θ possible $\overset{\text{means}}{\Leftrightarrow}$ O, Ω possible ✓ (and vice versa)

② Θ possible $\Rightarrow$ o, ω α (no chance of them)

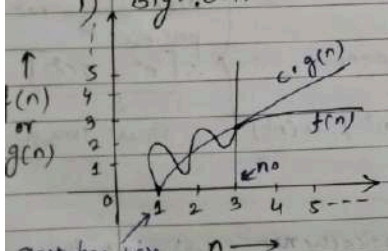
③ o possible $\Rightarrow$ O possible

④ O possible $\Rightarrow$ ~~ω~~ (small-oh may be, may not be)

⑤ ~~ω possible~~ ~~$\Rightarrow$~~

---

1) Big-oh.



Start from here ∵ n is integer ≥ 1 which is input to f(n) or g(n)

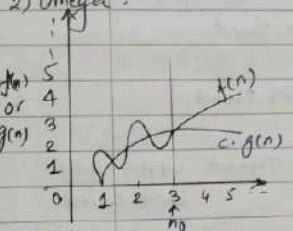
$f(n) = O(g(n))$

$\Updownarrow$

$f(n) \leq c \cdot g(n)$, $\forall n$, $n \geq n_0$.

---

2) Omega.



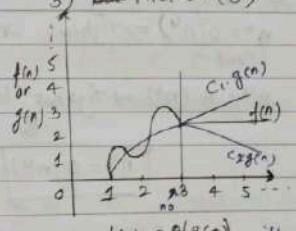
$f(n) = \Omega(g(n))$

$\Updownarrow$

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

$\forall n$, $n \geq n_0$

3) ~~Theta~~ Theta. (Θ)



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

① $f(n) \leq c_1 g(n)$ $\forall n, n \geq n_0$

② $f(n) \geq c_2 g(n)$

(greater than (or) equal to)

---

1) Big-oh ($f(n) \leq c \cdot g(n)$)

$n^2 = \begin{cases} O(n^2) \\ = O(n^3) \\ = O(n^{10}) \\ = O(n)\ \alpha \end{cases}$ → Tightest Upper Bound (∵ all upper bounds, it is equal to actual answer)

⇒ Upper Bounds (In Big-oh notation right side greater or equal)

$A = O(B)$

B: Upper Bound (Tightest / Not tight)

A: equal

Tightest UB / Not tight UB

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

$n^5 = \begin{cases} \Omega(n^5) \\ = \Omega(n^3) \\ = \Omega(n) \\ = \Omega(1) \\ = \Omega(n^{10})\ \alpha \end{cases}$ → Tightest lower Bound

⇒ lower Bounds (Right hand side always less or equal)

$A = \Omega(B)$

B: lower Bound (Tightest / Not tight)

A: equal

Tightest LB / Not tight LB

3) Theta

$n^4 = O(n^4) \Rightarrow$ Tightest Upper Bound.

and

$n^4 = \Omega(n^4) \Rightarrow$ Tightest lower Bound.

$\Downarrow$

$\boxed{n^4 = \Theta(n^4)} \Rightarrow$ Tightest Upper and lower Bound.

$\therefore A = \Theta(B)$

Both Tightest Upper & Tightest lower Bound.

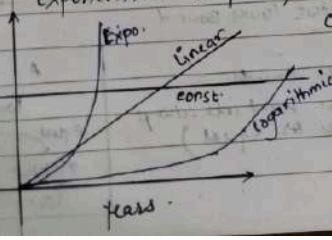
but still true

Increasing complexity

* **Complexity Classes** → both time & space.

Decreasing $f^n = \frac{1}{n}, \frac{1}{n^2}, \frac{1}{\log(n)} \cdots$ $\boxed{\frac{1}{n} = O(1)}$

1. Constant complexity $= O(1)$
2. Logarithmic Complexity $= O(\log n)$
3. Linear Complexity $= O(n)$
4. Quadratic Complexity $= O(n^2)$
5. Cubic complexity $= O(n^3)$
6. Polynomial complexity $= O(n^c)$ c is constant >0
   $\Rightarrow n^2, n^2, n^3, n^{0.5}, n^{0.00001}, n^{25} \cdots$
7. Exponential Complexity $= 2^n, 3^n, 7^n \cdots c^n$
   c is constant, $c > 1$



Expo.	Poly.
c^n	n^c
$c>1$	$c>0$

Increasing Complexity →

* $O(1) < O(\log(\log n)) < O(\log n) < O(\sqrt{n}) < O(n)$

* $O(n) < O(n \log n) < O(n\sqrt{n}) < O(n^2)$

* Inc. complexity ↓
- $O(\sqrt{n}) \longrightarrow n^{1/2} = n^{0.5}$ $(n^{0.5} < n^1)$
- $O(n)$
- $O(n\sqrt{n}) \longrightarrow n(n)^{1/2} = n^{3/2} = n^{1.5}$ $(n^{1.5} < n^2)$
- $O(n^2)$

* $n^{3/2} \Rightarrow n^{1.5} = n^1 \cdot n^{0.5} = n\sqrt{n}$ ($\because \log n$ similarly $\sqrt{n}$ $\Rightarrow$ $n \cdot \log n$, $n \cdot \sqrt{n}$, $n \cdot n$ ↓ Inc. comp.)

8. $2^n < n^n \Rightarrow (2 \cdot 2 \cdot 2 \cdots 2 < n \cdot n \cdot n \cdots n)$
   $2^n < n! \Rightarrow (2 \cdot 2 \cdot 2 \cdots 2 < n \cdot n-1 \cdot n-2 \cdots 2 \cdot 1)$
   $n! < n^n \Rightarrow (n \cdot n-1 \cdot n-2 \cdots 1 < n \cdot n \cdot n \cdots n)$

9. Sterling Approximation

$\Rightarrow \boxed{\log(n!) = n \log n}$

Now, we already know $n! < n^n$

taking log. $\Rightarrow \log(n!) < \log(n^n)$

$n \log n = O(n \log n)$ ← by Sterling approx^n

$\boxed{\log(n!) = O(\log n^n)}$ ← correct.

$n! = O(n^n)$ ✗ wrong

10. $\log_2 n > \log_3 n$ (smaller base, means larger value)

✓ Mathematically $\log_2 n$ is bigger than $\log_3 n$

But ✓ Asymptotically ⇒ $\log_3 n = \frac{\log_2 n}{\log_2 3} = \frac{\log_2 n}{1.5}$

∴ There is only constant difference b/w $\log_2 n$ and $\log_3 n \Rightarrow \left(\frac{\log_2 n}{1.5}\right)$

$\therefore \boxed{\log_2 n = \Theta(\log_3 n)}$ ⇒ Asymptotically equal

because all different base of $\log n$ differ by only constant.

11. $2^n < 3^n$

$2^n < (2 \times 1.5)^n$

$2^n < 2^n \times 1.5^n$

$1 < (1.5)^n \Rightarrow 2^n = \Theta(3^n)$ ∝ wrong

(small o pass means big-oh also pass) ⇒ $2^n = O(3^n)$ ✓ also $\therefore 2^n = o(3^n)$ ✓ but

because here there is not constant difference but there is function difference

∴ we cannot write $2^n = \Theta(3^n)$ ∝.

$\log_2 n$ & $\log_3 n$ differ by only constant

∴ we can write $\log_2 n = \Theta(\log_3 n)$

12. $n > \log n$ taking log on both sides

$n > (\log n)^2 \Rightarrow \log n > 2 \log n$

$n > (\log n)^3 \Rightarrow \log n > 3 \log n$

$n > (\log n)^{10000} \Rightarrow \log n > 10000 \log n$ } constant difference can be neglected.

$n < (\log n)^{\log n} \Rightarrow \log n = \log n \cdot \log(\log n)$

$1 < \log(\log n)$ } function difference can not be neglected.

$n < (\log n)^n$

13. $2^n < n^n$ but $\boxed{2^n > n^{\log n}}$

$n! < n^n$ ⇓ because

$2^n < n!$ $\log(2^n) > \log(n^{\log n})$ - (Apply log)

$n > (\log n)^2$ ⇐

14. $2^{2n} \overset{?}{} 2^n$ Which is bigger?

Don't Apply log blindly ⇒ First eliminate common thing.

$\therefore 2^n \cdot 2^n \overset{?}{} 2^n$

$\Rightarrow 2^n \_ 1$ $\Rightarrow \boxed{2^{2n} > 2^n}$

Now apply log ⇒ $\log(2^n) \_ \log(1)$

$n \geq 0$ ✓

Que) Check True / False

a) $1000\, n\log n = O\left(\frac{n\log n}{1000}\right)$

Soln $1000\, n\log n \le c \cdot \frac{n\log n}{1000}$

$(1000)^2\, n\log n \le c \cdot n\log n$

Here in place of c we can take $(1000)^3$ then above condn will be satisfied

∴ we can say $1000\, n\log n = O\left(\frac{n\log n}{1000}\right) \Rightarrow$ True

Method 2 :- $1000\, n\log n = O\left(\frac{n\log n}{1000}\right)$

Both side only constant difference ∴ we can neglect it $\Rightarrow$ $n\log n = O(n\log n) \Rightarrow$ True

* we can also prove that $1000 \cdot n\log n = \Omega\left(\frac{n\log n}{1000}\right)$

$1000\, n\log n \ge c \cdot \frac{n\log n}{1000}$

$(1000)^2\, n\log n \ge c \cdot n\log n$

Take c = 1 then above condn is satisfied

∴ we can say $1000\, n\log = \Omega\left(\frac{n\log n}{1000}\right) \Rightarrow$ True

Note :- Since it is Big-oh (O) also Omega (Ω)
It is also Theta (Θ)
Now, Since it is Theta, It is (not) small-oh (o)
and small omega (ω)

b) $\log(n) = O(\sqrt{n})$

$\log n \le c \cdot \sqrt{n}$

and we already know $\log n < \sqrt{n}$ ∴ Above condn is true

∴ $\log n = O(\sqrt{n}) \Rightarrow$ True, ($\le$ pass) } ∴ means O fail ∴ equal gone only < left

Now $\log n = \Omega(\sqrt{n}) \Rightarrow$ False ($\ge$ fail) }

actually it is $\log n = o(\sqrt{n})$
where $\log n < c \cdot \sqrt{n}$
(small oh pass)

c) If $0 < x < y$ then $n^x = O(n^y)$

Let x = 5 & y = 7

∴ $n^5 = O(n^7) \Rightarrow$ True = Big-oh pass

But $n^5 = \Omega(n^7)$
$n^5 \ne \Omega(n^5 \cdot n^2)$
$1 = \Omega(n^2)$ ∝ $\Rightarrow$ False, Omega fail. } means Theta (Θ) fail $\Rightarrow$ means equal gone $\Rightarrow$ It is small-oh.

d) $2^n = O(n^c)$ where c is constant > 0 $\Rightarrow$ False
2^n is exponential which is bigger than n^c which is polynomial.

e) $2^{2n} = O(2^n) \Rightarrow$ False

$2^{2n} \le c \cdot 2^n$

$2^n \cdot 2^n \le c \cdot 2^n$ $\Rightarrow$ Here c is constant ∴ This condn we are not able to prove it true ∴ Big-oh fail means small-oh also fail. ∴ O also fail.

∴ It is small omega (ω) $\Rightarrow$ $2^{2n} > 2^n$

f) $2^{n+1} = O(2^n) \rightarrow$ True

$2 \cdot 2^n \le c \cdot 2^n \Rightarrow$ can manage it

For constant only difference c

∴ Big-oh (O), Omega (Ω) both possible
∴ Theta (Θ)

g) $2^n = O(3^n)$ ⇒ False.

Big-oh ⇒ $2^n \leq 3^n$ ✓

Omega ⇒ $2^n \geq 3^n$ ✗ means Theta also Fail.

⇒ means equal gone ∴ actually its $2^n < 3^n$

⇒ Small-oh ✓

h) $\frac{4^n}{2^n} = O(2^n)$

⇒ $\left(\frac{4}{2}\right)^n$ ⇒ $2^n = O(2^n)$ ⇒ means $2^n = O(2^n)$ → True

** NOTE: Apply log ⇒

2^n	3^n
$\log(2^n)$	$\log(3^n)$
n	n ✗

But this is wrong method.

*DONT Apply log BLINDLY.

*first simplify with one in terms of another

2^n	3^n
2^n	$(2 \times 1.5)^n$
2^n	$2^n \times 1.5^n$

$1 < 1.5^n$ ✓

$\therefore 2^n = O(3^n)$

i) $\frac{1}{n} = O(1)$ → True. O ✓, Ω ✗, ∴ Θ ✗, ⇒ o ✓

j) $\frac{1}{n} = O(1)$ ⇒ False

$\therefore \frac{1}{n} \geq c \cdot 1$ ⇒ not possible as c is constant

∴ Ω fail ∴ Θ fail.

(↳ Right side person greater than n times ∴ They can never be equal.)

k) $n^2 \cdot 64^{\log_2 n} = O(n^{10})$

$= n^2 \cdot n^{\log_2 64} = O(n^{10})$

$\Rightarrow n^2 \cdot n^6 = O(n^{10})$

$\Rightarrow n^8 = O(n^{10})$ → True O ✓, Ω ✗ ⇒ Θ fail

+ Formula :- $a^{\log_c b} = b^{\log_c a}$

l) $128^{\log_2 n} \cdot 32^{\log_2 n} = O(n^{12})$

$\Rightarrow n^{\log_2 128} \cdot n^{\log_2 32} = O(n^{12})$

$\Rightarrow n^7 \cdot n^5 = O(n^{12})$

$\Rightarrow n^{12} = O(n^{12})$ → True ✓

Q) Check True or False.

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

Soln • we know $f(n)$ is a +ve f^n but we dont know it is increasing f^n or decreasing f^n.

① Increasing f^n

Let $f(n) = x$

$\therefore f(n) = O((f(n))^2)$

$x = O(x^2)$

True

② Decreasing f^n

Let $f(n) = 1/x$

$\therefore f(n) = O((f(n))^2)$

$\frac{1}{x} = O\left(\frac{1}{x^2}\right)$

False.

Considering Both Cases (True n False) = False.

$\therefore f(n) = O((f(n))^2)$ ⇒ FALSE for Dec. f^n

also ⇒ $f(n) = \Omega((f(n))^2)$ ⇒ False for Inc. f^n.

∴ These two are NOT comparable.

b) $f(n) = O\left(f\left(\frac{n}{2}\right)\right)$

⇒ Let $f(n) = n$ ⇒ $n = O\left(\frac{n}{2}\right)$

Here only constant difference there so we can neglect it ⇒ $n = O(n)$

⇒ $n = O(n)$

c) $f(n) = O\left(f\left(\frac{n}{2}\right)\right)$ ⇒ False.

Let ~~$f(n) = x^2$~~ ⇒ $x^2 = O\left(\left(\frac{x}{2}\right)^2\right)$

~~$\therefore f\left(\frac{n}{2}\right) = \left(\frac{x}{2}\right)^2$~~ ⇒ $x^2 = O\left(\frac{x^2}{4}\right)$

Let $f(n) = 2^n$ ⇒ $f(n) = O\left(f\left(\frac{n}{2}\right)\right)$

$\therefore f(n/2) = 2^{n/2}$ ⇒ $2^n = O\left(2^{n/2}\right)$

$2^{n/2} \cdot 2^{n/2} = O\left(2^{n/2}\right)$

$2^{n/2} = O(1)$

$2^{n/2} \leq c \cdot 1$ ⇒ False

To prove wrong, one example also enough ∴ here $f(n) = 2^n$ instead of x^2

2) Check True / False.

a) If $f(n) = O(g(n))$ then $g(n) = O(f(n))$

Sol^n

- Sometimes this passes as $n^2 = O(n^2)$ (f(n) = g(n))
- But even if we found one case where it fails then it is False.

∴ Let $f(n) = n^2$ & $g(n) = n^3$

$\therefore f(n) = O(g(n))$ ✓ but $g(n) = O(f(n))$ ✗

∴ Answer is FALSE

b) If $f(n) = O(g(n))$ then $g(n) = O(f(n))$ ⇒ TRUE

* Note :- Big-Oh (O) does not follow commutative property i.e if $A = O(B)$ then $B = O(A)$ ✗.

Theta (Θ) follows commutative property (Symmetric) i.e if $A = \Theta(B)$ then $B = \Theta(A)$ ✓

c) If $f(n) = O(g(n))$ & $g(n) = O(h(n))$ then $f(n) = O(h(n))$ ⇒ True.

Transitive property ✓ if $a \leq b \leq c$ then $a \leq c$

Let $a = n$, $b = n^2$, $c = n^4$

Big-Oh ⇒ $a \leq b \leq c$ = $n \leq n^2 \leq n^4$

then $a \leq c$ = $n \leq n^4$ ✓ Big-Oh pass

Omega ⇒ $a \geq b \geq c$ = $n^4 \geq n^2 \geq n$

then $a \geq c$ ⇒ $n^4 \geq n$ ✓ omega pass.

∴ Theta also pass.

Here small-oh & small omega also pass ✓.

* Note :- Here above question is not direct. They are conditional statements ∴ Theta pass then also small-oh and small omega pass.

But for direct statements, if theta pass then there is no chance that small-oh & small omega passes.

Check True / False.

a) If $f(n) = O(f(n))$ → True (Reflexive Property)
O, Ω, Θ ✓pass , o (small oh), ω ✗fail
If O pass then no chance of o, ω

b) If $f(n) = O(g(n))$ then
$h(n) \cdot f(n) = O(h(n) \cdot g(n))$
If $n \le n^3$ then
$n^2 \cdot n \le n^2 \cdot n^3$ ⇒ True

This satisfies for Ω also pass, ∴ Θ also pass
Also this satisfies for small oh (o) and ω (small omega) ✓
* This Question is conditional statement
∴ Θ also pass then also o, ω also pass.

c) If $f(n) = O(g(n))$ then
$2^{f(n)} = O(2^{g(n)})$

Let $f(n) = 2n$ $g(n) = n$
Here we can say $f(n) = O(g(n))$
⇒ $2n = O(n)$
∴ $2n \le \underset{\downarrow}{c} \cdot n$ ✓ satisfied
2

BUT $2^{f(n)} = O(2^{g(n)})$
$2^{2n} = O(2^n)$ ∴ $2^{f(n)} = O(2^{g(n)})$
⇒ $2^{2n} \le c \cdot 2^n$ ✗ Not satisfied False
$2^n \cdot 2^n \le \underset{\downarrow}{c} \cdot 2^n$
c cannot help ∵ it is constant

∵ $2^n \cdot 2^n \le c \cdot 2^n$ is not satisfied, we cannot say $2^{2n} = O(2^n)$ ∴ O (Big-oh) fail. ∴ Theta (Θ) fail
Ω (omega) fail.

∵ Theta Fail ⇒ small-oh may pass.
Let $f(n) = n$, $g(n) = n^2$ → (∵ for small-oh right side bigger)
$n < c \cdot n^2$ satisfies for $c = 1$ because of function difference and not worst difference
$\frac{f(n)}{g(n)} = o(n^2)$ ⇒ small-oh pass ✓
Similarly, we can prove ω pass ✓

* Imp Note :-
When we say "pass", everyone should pass. Even if atleast one case fail ⇒ we declare it as fail

Que) If $T_1(n) = O(f(n))$ && $T_2(n) = O(f(n))$
Check following statements is True / False

1. $T_1(n) = O(T_2(n))$ ⇒ False → $n^3 = O(n^4)$ && $n^3 = O(n^5)$ then $n^3 = O(n^2)$ ✗
2. $T_1(n) = \Omega(T_2(n))$ ⇒ False → $n^3 = \Omega(n^2)$ && $n^3 = \Omega(n^3)$ then $n^3 = \Omega(n^4)$ ✗
3. $T_1(n) = \Theta(T_2(n))$ ⇒ False
4. None

Que) If $T_1(n) = \Theta(f(n))$ && $T_2(n) = \Theta(f(n))$ check T/F
1. $T_1(n) = O(T_2(n))$: True
2. $T_1(n) = \Omega(T_2(n))$: True
3. $T_1(n) = \Theta(T_2(n))$ → True
4. None.

Here $n^3 = O(n^3)$ ✓
i.e let $a = n^3$, $b = n^3$
then we can say $a = b$
∴ Θ satisfies, O and Ω also satisfies.

Que) $f(n) = \log^*(\log n)$ $\quad g(n) = \log(\log^* n)$
What is rel^n b/w them?

Sol^n Meaning of $\log_2^* n$ ⇒ The no. of times you apply log so that n becomes equal to 1.

(i) Sol^n:- Let $n = 2^{2^{2^{2^2}}}$ } 5 times

$\log_2(2^{2^{2^{2^2}}}) = 2^{2^{2^2}} \Rightarrow \log_2(2^{2^{2^2}}) = 2^{2^2} \Rightarrow \log_2(2^{2^2}) = 2^2$

$\log_2(2^2) = 2 \Rightarrow \log_2(2) = 1$

$\therefore \log_2^* n = 5$

(ii) $n = 2^{2^{\cdot^{\cdot^{\cdot^2}}}}$ } 1,00,000 $\Rightarrow \log_2^* n = 1,00,000$

$\log_2 n = 2^{2^{\cdot^{\cdot^{\cdot^2}}}}$ } 99,999

↓ Apply log.

$\log_2(\log_2^* n) = 17 \Leftrightarrow g(n)$

$\log_2^*(\log_2 n) = 99,999 \Rightarrow f(n)$

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

Que) If $f(n) = O(d(n))$ && $g(n) = O(e(n))$ then $f(n) + g(n) =$ ____ ?

Sol^n $f(n) + g(n) = O(d(n) + e(n))$

(or)

$O(\max(d(n), e(n)))$

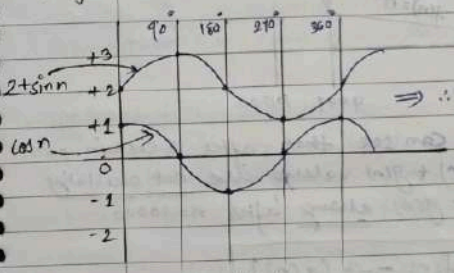
Que) If $f(n) = O(d(n))$ && $g(n) = O(e(n))$ then $f(n) \cdot g(n) = ?$

Sol^n

$f(n) \cdot g(n) = O(d(n) \cdot e(n))$

Que) Rel^n b/w the following 2-functions

(i) $f(n) = n^{2+\sin(n)}$ $\quad g(n) = n^{\cos n}$

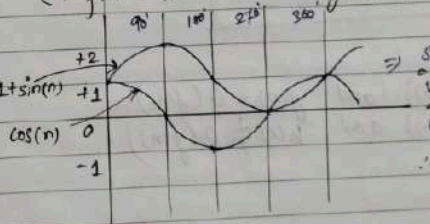


$\Rightarrow \therefore f(n) = \Omega(g(n))$

(or)

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

(ii) $f(n) = n^{1+\sin(n)}$ $\quad g(n) = n^{\cos(n)}$



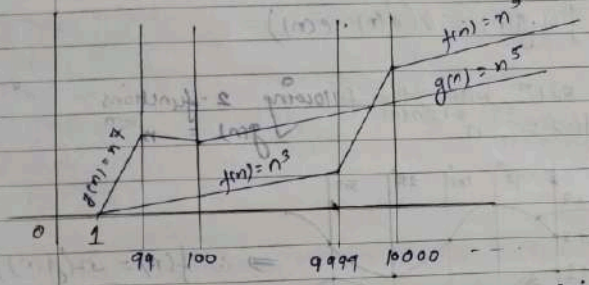
⇒ So, here sometimes $f(n)$ is bigger, sometimes $g(n)$ is bigger

∴ These two fns are not comparable.

Que) Find relⁿ b/w $f(n)$ & $g(n)$

$$f(n) = \begin{cases} n^3 & 0 < n < 10000 \\ n^9 & n \geq 10000 \end{cases}$$

$$g(n) = \begin{cases} n^7 & 0 < n < 100 \\ n^5 & n \geq 100 \end{cases}$$



Here we can see that after a certain point 10000, $f(n)$ & $g(n)$ values are not oscillating but $f(n) > g(n)$ always after $n = 10000$

$$\therefore f(n) = \Omega(g(n)), \quad n_0 = 10000$$

Que) Let us assume

① $f(n) = O(g(n))$ and $g(n) \neq O(f(n))$

② $g(n) = O(h(n))$ and $h(n) \neq O(g(n))$

then

options:-

a) $f(n) = \Theta(h(n))$

b) $h(n) \cdot f(n) = O(f(n) \cdot g(n))$

c) $f(n) \cdot g(n) = O(g(n) \cdot h(n))$

d) $f(n) + g(n) = \Theta(h(n))$

Sol^n given that $f(n) = O(g(n))$ and $g(n) = O(f(n))$; based on

① $\therefore$ we can conclude that $\boxed{f(n) = g(n)} \rightarrow f = \Theta$

Also, given that $g(n) = O(h(n)) \Rightarrow g(n) \leq h(n)$

② and $h(n) \neq O(g(n)) \Rightarrow h(n) \neq g(n)$

i.e $h(n) \neq g(n)$ also $h(n) \not\leq g(n)$ (equal gone because of this)

$\therefore$ we can conclude that $\boxed{g(n) < h(n)}$

$g = o(h)$

$\therefore$ From above we can conclude $\Rightarrow \boxed{f(n) = g(n) < h(n)}$

$\therefore$ options a) $f(n) = \Theta(h(n)) \rightarrow$ False

b) $h(n) \cdot f(n) = O(f(n) \cdot g(n))$

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

$\Rightarrow h(n) \leq c \cdot g(n) \rightarrow$ False

c) $f(n) \cdot g(n) = O(g(n) \cdot h(n))$

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

$\Rightarrow f(n) \leq c \cdot h(n) \rightarrow$ True

d) $f(n) + g(n) = \Theta(h(n))$

$\Rightarrow$ False

Recursion

① * A function calling itself to solve a problem is known as recursion.

Eg:-
```
A()
{
  ...
  A();
  ...
}
```

Eg:- Fact(n) = n x Fact(n-1)

Fact(6) = 6 x Fact(5) → 6x120 = 720
5 x Fact(4) → 5x24 = 120
4 x Fact(3) → 4x6 = 24
3 x Fact(2) → 3x2 = 6

② * Recursion is solving bigger problems in terms of smaller problem is known as recursion.

Note :-
When Fact(6) is called, it calls Fact(5) and Fact(6) waits for Fact(5) to complete its execution.
Fact(5) inturn calls Fact(4) and waits for it to complete its execution... and so on.
- This waiting should be in correct order
∴ A specific data structure "stack" is needed.
As soon as Fact(6) called, it is pushed into stack.
∴ Function call is a push operation.
& non recursive

③ To execute a recursive program, we use stack data structure.

* In case of non recursive program, stack size is based on no. of fn calls which are less (constant)
∴ constant stack size is required.

* But, In case of recursive program, stack size is based on no. of inputs. (eg:- Fact(6) requires stack size = 6 because 6 times fn call is there)
So, here stack size is not constant and stack space is required acc. to recursive program (input)
therefore here, we have to consider space complexity in case of recursion.

* Recursive Program :- Factorial.

④ In every recursive program, we have to write a termination condition which will end the program. This terminat" cond" should be written at the start of the program only. (otherwise it will give run time error. (Stack overflows))

```
fact(n)
{
  if(n <= 1)  ---> termination cond".
    return(1);
  else
  return(n x fact(n-1));
}
```

∴ A recursive prog. contains 2 parts :-
① if part : contains termination cond".
② else part : contains major logic.

* Recursive Program

```
fact(n)
{
  if(n<=1) return(1);
  else
  return(n*fact(n-1));
}
```

→ Function call inside function
→ No. of f^n calls = n
→ stack space = n ⇒ O(n) (space comp.)

* Non Recursive Program

```
fact(n)
{
  s=1
  for(i=1; i<=n; i++)
  {
    s=s*i;
  }
  return(s);
}
```

→ No function call inside function
→ No. of f^n calls = 1 (fact)
→ stack space = 1 ⇒ O(1)

(5) ∴ For every recursion, Non recursion possible and vice-versa.

(6) Comparing Recursive and Non recursive program, both will take same Time (same time complexity) because of same logic. Recursive prog. will take more space because of more f^n call.

(7) The amount of stack space took while running a Recursive program is known as 'depth of the stack'

(8) While Recursive Prog. is running, from one f^n call to another f^n call only parameter value changes and not no. of parameter, name of parameter, code / entire program. Only parameter value changes

Note :- If you write a recursive program in the form of a mathematical relation, then it is called as Recurrence relation.

* $f(n) = \begin{cases} 1 & \text{if } n \le 1 \\ n \times fact(n-1) & \text{if } n > 1 \end{cases}$ → Recurrance rel^n to calculate value

* Now, Recurrance rel^n to cal. Time Complexity Here, we have to take care of 'loops'

T(n)

```
fact(n)
{
  if(n<=1) return(1)
  else
  return(n*fact(n-1))
}
```

T(n-1)

$$T(n) = \begin{cases} O(1) & \text{if } n \le 1 \\ T(n-1) + c & \text{if } n > 1 \end{cases}$$

↓ Recurrance rel^n to calculate time.

Thus, for what purpose we are writing recurrance rel^n, we will get that. If we write recurrence rel^n for value, we get value of the program. If we write recurrence rel^n for time, we get time comp for that program.

* Therefore, one recursive program has many recurrence rel^n.

Qu) Write a Recursive program and Recursive Relation to find mul(m,n) where m, n ≥ 1

Sol^n i/p : 5, 2 o/p = 10

Logic → $m \times n$ = adding m, n times (or) adding n, m times

$\therefore 5 \times 2$ = $\underbrace{5+5}_{2 \text{ times}}$ (or) $\underbrace{2+2+2+2+2}_{5 \text{ times}}$

$\therefore$ **mul(m,n) = m + mul(m,n-1)**

Eg: mul(2,5) = 2 + mul(2,4)

2 + mul(2,3) → 6+2=8

2 + mul(2,2) → 4+2=6

2 + mul(2,1) → 2+2=4

2 + mul(2,0) → 2+0=2

0

8+2=10

∴ Recursive Program.

```
mul(m, n)
{
  if(m==0||n==0)
    return (0);
  else
    return (m+mul(m,n-1));
}
```

* Recurrence Relation (for finding value)

* write as it is by seeing program. Little change also may affect entire value.

$\therefore mul(m,n) = \begin{cases} 0 & \text{if } m=0 \text{ (or) } n=0 \\ m + mul(m,n-1) & \text{otherwise} \end{cases}$

⇓

If we solve this, we will get value

* Recurrence Rel^n (for finding Time complexity)

* For time comp., think about loops.

T(m,n)

```
mul(m,n);
{
  if (m==0||n==0)
    return (0)
  else
    return (m + mul(m,n-1))
}
```

T(m,n-1)

$\Rightarrow T(m,n) = \begin{cases} O(1) & \text{if } m=0 \text{ or } n=0 \\ T(m,n-1) + c & \text{otherwise} \end{cases}$

for recursion; other than recursion (∵ there are no loops in else part write constant)

* Recurrence relation (for Addition)

while writing rec. rel^n for Time complexity we concentrated on loops. But to find recurrence rel^n for Add^n, we will concentrate on Addition operator

$\therefore A(m,n) = \begin{cases} 0 & \text{if } m=0 \text{ or } n=0 \\ A(m,n-1) + 1 & \text{otherwise} \end{cases}$ } (no. of add^n operator in 'if' part of code)

for recursion; 1 time Add^n operator

⇓

If we solve this, we will get no. of Add^n operators in our code.

Que) Write Recurrence Rel^n & Recursive Program for n^{th} fibonacci number.

n	0	1	2	3	4	5	6	7	8	9	10
fib(n)	0	1	1	2	3	5	8	13	21	34	55

$fib(100) = fib(99) + fib(98) \Rightarrow \boxed{fib(n) = fib(n-1) + fib(n-2)}$

Recursive Program
fib(n) ⇒ T(n)

```
if(n==0 || n==1)
    return (n);
else
    return (fib(n-1) + fib(n-2));
```

T(n-1) T(n-2)

* Recurrence Rel^n (for value)

write as it is

$$V(n) = \begin{cases} n & \text{if } n=0 \text{ or } n=1 \\ V(n-1)+V(n-2) & \text{other} \end{cases}$$

* Recurrence Rel^n (for Time)

no loop in if part

$$T(n) = \begin{cases} O(1) & \text{if } n=0 \text{ or } n=1 \\ T(n-1)+T(n-2)+O(1) & \text{if } n>1 \end{cases}$$

↑ in else part, no loop there ∴ constant time (C)

* Recurrence rel^n (for Addition)

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

For recursion; ↑ one time add^n operator

Recurrence Relation Solving

methods :- ① Substitution Method.
② Recursive Tree Method.
③ Master Theorem

① Substitution Method. → substituting the given f^n again and again till termination cond^n happens.

Eg:- $T(n) = \begin{cases} 1 & \text{if } n=1 \\ T(n-1)+n & \text{if } n>1 \end{cases}$

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

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

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

↓ 100 times

$= T(n-100) + n-99 + n-98 \cdots + n-1 + n$

↓ k times

$T(n) = T(n-k) + n-(k-1) + n-(k-2) + \cdots n-1+n$

Here, we called recursive f^n k times ⇒ This is stack space $\boxed{k = n-1} = O(n)$

Now, Here termination Cond^n is $T(1) = 1$ ⇒ ∴ Put $n-k=1$ ⇒ $(k = n-1)$

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

$= T(1) + 2 + 3 + 4 + \cdots n-1+n$

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

$T(n) = \frac{n(n+1)}{2} \Rightarrow \boxed{T(n) = O(n^2)}$

Eg 2): $T(n) = \begin{cases} 1 & \text{if } n=0 \\ T(n-1) \times n & \text{if } n>0 \end{cases}$ Solve by substitution.

Sol^n $T(n) = T(n-1) \times n$

$= T(n-2) \times n-1 \times n$

$= T(n-3) \times n-2 \times n-1 \times n$

↓ k times

$T(n-k) * n-(k-1) * n-(k-2) \cdots n$

Termination cond^n ⇒ $n-1c=0$ ∴ $T(0)=1$

$k=n$ ⟶ recursion f^n called k times

∴ stack space = $k=n$ = $O(n)$

∴ $T(n) = T(0) * n-(n-1) * n-(n-2) \cdots n$

$= T(0) * 1 * 2 * 3 \cdots * n$

$T(n) = 1 \times 1 \times 2 \times 3 \cdots n$

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

$= \Omega(2^n)$

$= \Theta(n!)$

Eg 3):

$$T(n) = \begin{cases} 1 & \text{if } n=0 \\ T(n-2)+n^2 & \text{if } n>0 \end{cases}$$

Sol^n $T(n) = T(n-2) + n^2$ — 1 time substitut^n

$T(n-4) + (n-2)^2 + n^2$ — 2 time

$T(n-6) + (n-4)^2 + (n-2)^2 + n^2$ — 3 time

↓ k times

$T(n) = T(n-2k) + (n-(2k-2))^2 + (n-(2k-4))^2 \cdots n^2$

Termination cond^n ⇒ $T(0)=1$ ⇒ put $n-2k=0 \Rightarrow 2k=n$

$\therefore k = n/2$

$T(n) = T(0) + 2^2 + 4^2 + 6^2 \cdots n^2$

$= 1 + [2^2 + 4^2 + 6^2 \cdots + n^2]$ → n/2 terms.

$= 1 + [(2\times1)^2 + (2\times2)^2 + (2\times3)^2 \cdots (2\times\frac{n}{2})^2]$

$= 1 + 2^2[1^2 + 2^2 + 3^2 + \cdots (n/2)^2]$

$T(n) = 1 + 2^2\left[\frac{(n/2)(n/2+1)(2(n/2)+1)}{6}\right] = O(n^3)$

Que) $T(n) = \begin{cases} 1 & \text{if } n=0 \\ T(n-2)+\log n & \text{if } n>0 \end{cases}$ → since we are always subtracting 2 ∴ no of levels = $\frac{n}{2}$

Sol^n $T(n) = T(n-2) + \log(n)$

$= T(n-4) + \log(n-2) + \log n$

$= T(n-6) + \log(n-4) + \log(n-2) + \log n$

↓ k times

$= T(n-2k) + \log(n-(2k-2)) + \log(n-(2k-4)) \cdots \log(n-0)$

Termination cond^n ⇒ $n-2k=0 \Rightarrow 2k=n$

$\therefore k = n/2$

$\therefore T(n) = T(0) + \log(2) + \log(4) \cdots \log(n)$

$T(n) = 1 + [\log 2 + \log 4 + \log 6 \cdots \log n]$

$= 1 + [\log(2\times4\times6\times8 \cdots n)]$ → n/2 terms.

$= 1 + [\log_2(2^{n/2} \times (1\times2\times3\times4 \cdots n/2))]$

$= 1 + [\log_2(2^{n/2}) + \log_2(1\times2\times3 \cdots n/2)]$

$= 1 + [n/2 + \log_2(n/2)!]$

$= 1 + [\frac{n}{2} + \log(n/2)^{n/2}]$ ⇒ ∵ $\log(n!) = \log(n^n)$ by Stirling approx^n

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

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

NOTE: usually $n! < n^n$ but by Stirling approx^n $\log(n!) = \log(n^n)$

Que 2) $T(n) = \begin{cases} 1 & \text{if } n=1 \\ T(n-1)+c & \text{if } n>1 \end{cases}$

Sol^n $T(n) = T(n-1) + c$
$= T(n-2) + c + c$
$= T(n-3) + c + c + c$
$\downarrow k$
$= T(n-k) + kc$

Here, everytime a non recursive term is coming with no difference than previous ∴ It will lead to no series

Terminat^n $n-k = 1 \Rightarrow k = n-1$

$\therefore T(n) = T(1) + (n-1)c = 1 + (n-1)c = O(n)$
$= \Theta(n)$
$= \Omega(n)$
$= O(n^5)$
$= \Omega(1)$

Que 3) $T(n) = \begin{cases} 1 & \text{if } n=1 \\ T(n/2)+n & \text{if } n>1 \end{cases}$

Sol^n $T(n) = T(n/2) + n$
$= T(n/4) + n/2 + n \Rightarrow T(n/2^2) + n/2^1 + n/2^0$
$= T(n/8) + n/4 + n/2 + n \Rightarrow T(n/2^3) + n/2^2 + n/2^1 + n/2^0$
$\downarrow k$
$= T(n/2^k) + n/2^{k-1} + n/2^{k-2} \cdots \frac{n}{2^1} + \frac{n}{2^0}$

Terminat^n $\Rightarrow \frac{n}{2^k} = 1 \Rightarrow 2^k = n \Rightarrow \boxed{k = \log_2 n}$

$\therefore T(n) = T(1) + \frac{n}{2^{\log n - 1}} + \frac{n}{2^{\log n - 2}} \cdots \frac{n}{2^1} + \frac{n}{2^0}$
$= 1 + n\left[\left(\frac{1}{2}\right)^{\log n - 1} + \left(\frac{1}{2}\right)^{\log n - 2} + \cdots \left(\frac{1}{2}\right)^1 + \left(\frac{1}{2}\right)^0\right]$
$= 1 + n\left[\left(\frac{1}{2}\right)^0 + \left(\frac{1}{2}\right)^1 + \left(\frac{1}{2}\right)^2 \cdots + \left(\frac{1}{2}\right)^{\log n - 2} + \left(\frac{1}{2}\right)^{\log n - 1}\right]$

Sum of n terms in a GP $= \frac{a(r^n - 1)}{r-1}, \quad r > 1$

Sum of n terms in GP $= \frac{a(1-r^n)}{1-r}, \quad r < 1$

$\therefore T(n) = 1 + n\left[\frac{1\left(1 - (1/2)^{\log n}\right)}{1 - 1/2}\right]$

$T(n) = 1 + 2n\left[1 - \frac{1}{2^{\log n}}\right]$

$T(n) = 1 + 2n\left[1 - \frac{1}{n^{\log_n 2}}\right] \rightarrow \left(\because a^{\log b} = b^{\log a}\right)$

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

For increasing GP, do normal way, no shortcut

* Short trick :- A decreasing GP ($r < 1$) $\Rightarrow$ complexity is $O(1)$

$\therefore T(n) = 1 + n\left[\left(\frac{1}{2}\right)^0 + \left(\frac{1}{2}\right)^1 + \left(\frac{1}{2}\right)^2 \cdots \left(\frac{1}{2}\right)^{\log n - 1}\right]$

$T(n) = 1 + n[O(1)] \Rightarrow \boxed{T(n) = O(n)}$

Que 4) $T(n) = \begin{cases} 1 & \text{if } n=1 \\ 8T(n/2) + n^2 & \text{if } n>1 \end{cases}$

Sol^n $T(n) = 8T(n/2) + n^2$
$T(n) = 8\left[8T(n/4) + (n/2)^2\right] + n^2 = 8^2 T\left(\frac{n}{2^2}\right) + 2n^2 + n^2$
$T(n) = 8^2\left[8T\left(\frac{n}{2^3}\right) + \left(\frac{n}{2^2}\right)^2\right] + 2n^2 + n^2$
$= 8^3 T\left(\frac{n}{2^3}\right) + 4n^2 + 2n^2 + n^2$
$\downarrow k$ times
$= 8^k T\left(\frac{n}{2^k}\right) + n^2\left[2^{k-1} + 2^{k-2} + \cdots + 2^1 + 2^0\right]$

$\frac{n}{2^k} = 1$
$n = 2^k$
$k = \log_2 n$

$= 8^{\log_2 n} T(1) + n^2\left[2^0 + 2^1 + 2^2 + \cdots 2^{\log n - 2} + 2^{\log n - 1}\right]$

Inc. GP with $\log n$ terms, $r > 1$

$= n^{\log_2 8} + n^2 \left[\frac{1(2^{\log_2 n} - 1)}{2-1} \right]$

$= n^3 + n^2 (n-1) = O(n^3)$

Qué) $T(n) = \begin{cases} 1 & \text{if } n=1 \\ 2T(n/2) + n\log n & \text{if } n>1 \end{cases}$

Soln $T(n) = 2T(n/2) + n\log n$

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

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

↓ k time

$T(n) = 2^k T\left(\frac{n}{2^k}\right) + n\log\left(\frac{n}{2^{k-1}}\right) + n\log\left(\frac{n}{2^{k-2}}\right) + \cdots + n\log\left(\frac{n}{2^0}\right)$

← write in reverse

$\frac{n}{2^k} = 1$ ⇒ $T(n) = n\,T(1) + n\left[\log\frac{n}{2^0} + \log\frac{n}{2^1} + \cdots + \log\frac{n}{2^{\log n - 1}}\right]$

$n = 2^k$

$k = \log_2 n$ $T(n) = n + n\left[(\log_2 n - \log_2 2^0) + (\log n - \log_2 2^1) + (\log n - \log_2 2^2) + \cdots + (\log_2 n - \log_2 2^{\log n - 1})\right]$

$= n + n\left[\underbrace{\log n + \log n + \cdots \log n}_{\log_2 n \text{ times}} - (\log 2^0 + \log 2^1 + \log 2^2 \cdots)\right]$

$= n + n\left[\log_2 n \times \log_2 n - \underbrace{(0+1+2+3\cdots+\log n - 1)}_{\log n \text{ terms}}\right]$

$= n + n\left[(\log_2 n)^2 - \frac{\log n(\log n + 1)}{2}\right]$

$= n + n\log n\left[\log n - \frac{\log n}{2} - \frac{1}{2}\right] = n + n\log n\left(\frac{\log n}{2} - \frac{1}{2}\right)$

$T(n) = n + \frac{n}{2}(\log n)^2 - \frac{n}{2}\log n$ ⇒ $T(n) = O\left[n(\log n)^2\right]$

Qué) $T(n) = \begin{cases} 2 & \text{if } n=2 \\ \sqrt{n}\,T(\sqrt{n}) + n & \text{if } n>2 \end{cases}$

Soln $T(n) = \sqrt{n}\,T(\sqrt{n}) + n = n^{1/2} \cdot T(n^{1/2}) + n$ — 1st level

Substitute and find $T(\sqrt{n})$ value

$T(n) = n^{1/2}\left[n^{1/2^2} T(n^{1/2^2}) + n^{1/2}\right] + n$ — 2nd level.

$= n^{3/2^2} T(n^{1/2^2}) + n + n$ ⇒ Here we got same recursion term even after substitute.

∴ It will not form any new terms

$T(n) = n^{3/2^2}\left[n^{1/2^3} T(n^{1/2^3}) + n^{1/2^2}\right] + n + n$

$= n^{7/2^3} T(n^{1/2^3}) + n + n + n$ — 3rd level.

↓ k times

$T(n) = n^{2^k - 1/2^k} T\left(\frac{1}{n^{2^k}}\right) + kn = n^{\frac{2^k}{2^k} - \frac{1}{2^k}} T(n^{1/2^k}) + kn$

$T(n) = \frac{n^1}{n^{1/2^k}} T(n^{1/2^k}) + kn$

Terminate ⇒ $n^{1/2^k} = 2$

$2^k = \log_2 n$

$k = \log_2(\log_2 n)$

$\therefore T(n) = \frac{n}{2} T(2) + \log_2(\log_2 n) \cdot n$

$T(n) = n + n \cdot \log(\log n) = O(n \cdot \log(\log n))$

Qué) $T(n) = \begin{cases} 1 & \text{if } n=1 \\ 2T(n-1) + n & \text{if } n>1 \end{cases}$

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

$T(n) = 2\left[2T(n-2) + n-1\right] + n$

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

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

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

$T(n) = 2^1\left[2T(n-3) + n-2\right] + 2n - 2 + n$

$= 2^3 T(n-3) + 2^2 n - 2^3 + 2n - 2 + n$

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

↓ k times

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

$\cdots - 2^2(n-2) + 2^1(n-1) + 2^0(n-0)$

Terminating cond^n $\Rightarrow n-k=1 \Rightarrow k = n-1$

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

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

↳ This is a AGP series.

$2^0, 2^1, 2^2, 2^3 \cdots 2^{n-1} \Rightarrow$ GP $(r=2)$

Solving AGP $(n-0), (n-1), (n-2), \cdots 2, 1 \Rightarrow$ AP

∴ For GP, $r=2$ ∴ multiply series by 2 & substract

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

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

$T(n) - 2T(n) = 2^0(n-0) - 2^1(1) - 2^2(1) - 2^3 - 2^4 \cdots - 2^{n-2} - 2^{n-1} - 2^n$

$-T(n) = n - \left[2^1 + 2^2 + 2^3 + \cdots 2^n\right]$

$-T(n) = n - \left[\frac{2(2^n-1)}{2-1}\right] \Rightarrow T(n) = 2(2^n-1) - n$

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

$= O(2^n)$

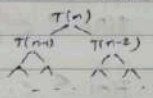
(2) Recursive Tree Method (Lecture -15)

Eg:- $T(n) = T(n-1) + T(n-2) + \text{constant}(C)$

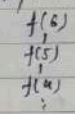
This recurrence rel^n is for fibonacci series Time complexity

Here, in each recursion, 2 function calls are there

∴ It will form a binary tree



But for factorial program, each recursion calls 1 function only. ∴ It will form a unary tree.

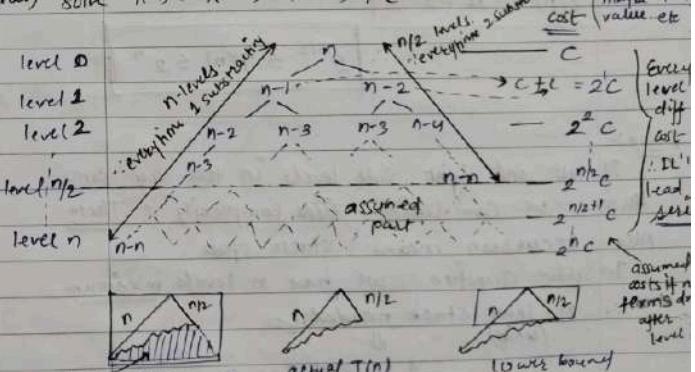


Note

** Recursive Tree method is preferred when there are more than one function calls in the recurrence rel^n.

Rece. Rel^n for fibonacci series Time complexity

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



lower Bound $\leq T(n) \leq$ upper Bound

$\therefore T(n) \leq 2^0 c + 2^1 c + 2^2 c + \dots + 2^n c$

↑ comparing actual answer (which is $T(n)$), right hand side is more (because of assumed part)

$\Rightarrow T(n) \leq c\left[2^0 + 2^1 + 2^2 + \dots + 2^n\right]$

$T(n) \leq c\left[\frac{1(2^n - 1)}{2-1}\right]$

$T(n) \leq c \cdot 2^n \Rightarrow \boxed{T(n) = O(2^n)}$

This is Time comp. of fibonacci. because recurrence rel^n was given for Time complexity

for lower bound :-

$T(n) \geq 2^0 c + 2^1 c + \dots + 2^{n/2} c$

↑ comparing actual answer ($T(n)$), right hand side is less.

$\Rightarrow T(n) \geq c \cdot \left[2^0 + 2^1 + 2^2 + \dots + 2^{n/2}\right]$

$\Rightarrow T(n) \geq c \cdot 2^{n/2}$

$\Rightarrow \boxed{T(n) = \Omega(2^{n/2})}$

$\therefore \boxed{2^{n/2} \leq T(n) \leq 2^n}$

* Note :-

If left and right side levels in tree are same, then we can write time complexity = Theta

Also, recursion means stack space.

∴ In above Question, we have n levels maximum

∴ n - level stack needed. (units)

⇓

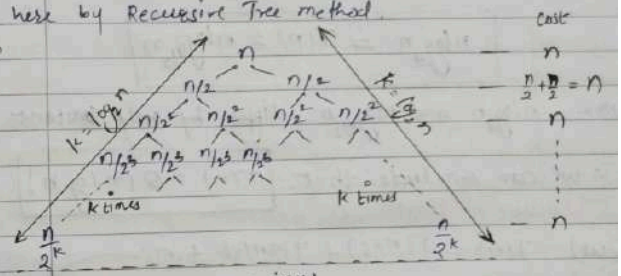
1 stack of size n

Que) $T(n) = T(n/2) + T(n/2) + n$ ← cost

we can do this by substitution method also

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

But, here by Recursive Tree method



$\therefore T(n) \leq n + n + \dots + n$ (log n times)

$= O(n \log n)$

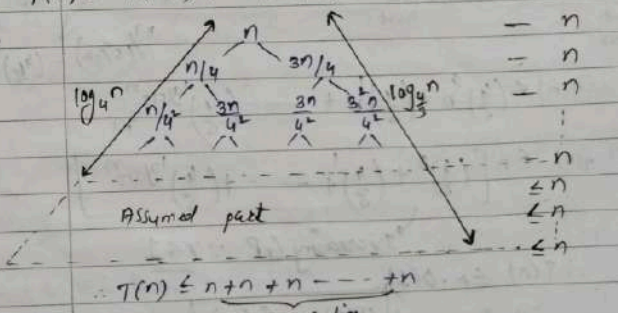
$T(n) \geq n + n + \dots + n$

$= \Omega(n \log n)$

$\therefore \boxed{T(n) = \Theta(n \log n)}$

Tree height = stack space = $O(\log n)$

Que) $T(n) = T(n/4) + T(3n/4) + n$



$\therefore T(n) \leq n + n + n + \dots + n$ ($\log_{4/3} n$ times)

$T(n) \leq n \log_{\frac{4}{3}} n = O(n \log_{4/3} n)$

For lower Bound ⇒ $T(n) \geq \underbrace{n+n+n--n}_{\log_4 n \text{ times}}$

$T(n) = \Omega(n \log_4 n)$

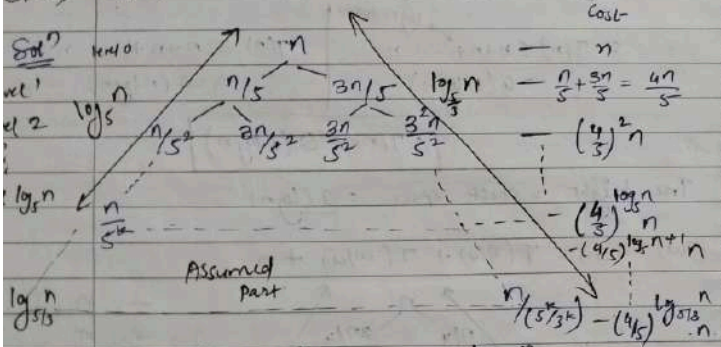
$\therefore \boxed{n\log_4 n \leq T(n) \leq n\log_{4/3} n}$

Now, $n\log_4 n$ and $n\log_{4/3} n$ differ by only constant

$\therefore$ we can conclude that $\boxed{T(n) = \Theta(n \log n)}$

(Que) $T(n) = T(n/5) + T(3n/5) + n$ → cost

Soln



$\therefore T(n) \leq \left(\frac{4}{5}\right)^0 n + \left(\frac{4}{5}\right)^1 n + \cdots + \left(\frac{4}{5}\right)^{\log_{5/3} n} n$

$T(n) \leq n\left[\left(\frac{4}{5}\right)^0 + \left(\frac{4}{5}\right)^1 + \cdots + \left(\frac{4}{5}\right)^{\log_{5/3} n}\right]$

↳ Decreasing GP $\because r < 1$

$\therefore T(n) \leq n \cdot O(1)$

$\Rightarrow \boxed{T(n) = O(n)}$

Lower Bound ⇒ $T(n) \geq n\left[(4/5)^0 + (4/5)^1 + (4/5)^2 \cdots (4/5)^{\log_5 n}\right]$

$T(n) \geq n \cdot O(1)$ ← Decreasing GP $\because r < 1$

$\boxed{T(n) = \Omega(n)}$

* $\because$ we can say that $\boxed{T(n) = \Theta(n)}$

* Note :- In this Question $T(n) = T(n/5) + T(3n/5)$

$T(n/5)$: always divide by 5 ∴ No of levels = $\log_5 n$

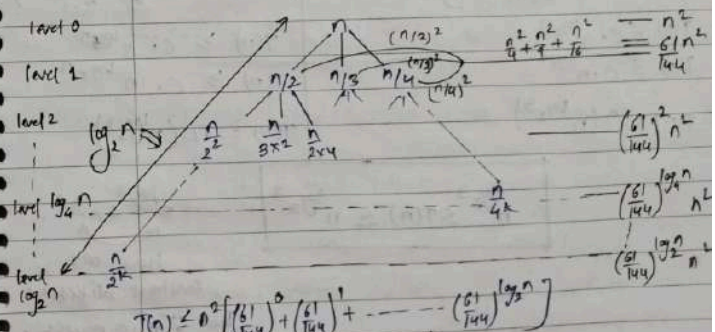
$T(3n/5)$: divide by 5/3 ∴ levels = $\log_{5/3} n$

Smaller base means larger value ∴ $\log_{5/3} n > \log_5 n$

* Max^m Height of tree = $\log_{5/3} n$ ← Stack space

(Que) $T(n) = T(n/2) + T(n/3) + T(n/4) + n^2$

↳ Here, It will form a ternary tree.



$T(n) \leq n^2\left[\left(\frac{61}{144}\right)^0 + \left(\frac{61}{144}\right)^1 + \cdots \left(\frac{61}{144}\right)^{\log_3 n}\right]$

$T(n) \leq n^2 [O(1)]$

$= O(n^2)$

↳ Decreasing GP $\because r < 1$

Lower Bound ⇒ $T(n) \geq n^2\left[\left(\frac{6}{144}\right)^0 + \left(\frac{6}{144}\right)^1 + \cdots \left(\frac{6}{144}\right)^{\log_4 n}\right]$

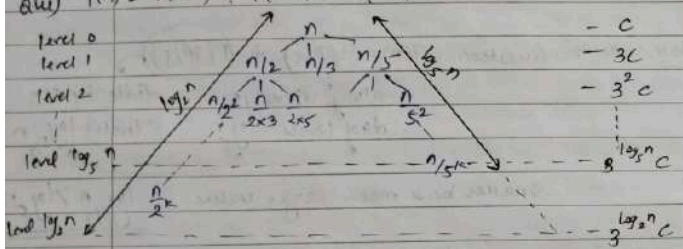
$\geq n^2[\Omega(1)]$

$T(n) = \Omega(n^2)$

∴ we can say $\boxed{T(n) = \Theta(n^2)}$

Stack Space ⇓ $O(\log_2 n)$

Q(iii) $T(n) = T(n/2) + T(n/3) + T(n/5) + c$



Upper Bound :- (increasing GP)

$T(n) \leq c\left[3^0 + 3^1 + 3^2 - - - - + 3^{\log_2 n}\right]$

$T(n) \leq c\left[\frac{1\cdot(3^{\log_2 n} - 1)}{3-1}\right]$

$T(n) \leq c \cdot 3^{\log_2 n}$

$T(n) \leq c \cdot n^{\log_2 3}$

$T(n) = O(n^{\log_2 3})$

Lower Bound :-

$T(n) \geq c\left[3^0 + 3^1 + - - 3^{\log_5 n}\right]$

$T(n) \geq c\cdot\left[\frac{1\cdot(3^{\log_5 n} - 1)}{3-1}\right]$

$T(n) \geq c \cdot 3^{\log_5 n}$

$T(n) \geq c \cdot n^{\log_5 3}$

$T(n) = \Omega(n^{\log_5 3})$

$\therefore n^{\log_5 3} \leq T(n) \leq n^{\log_2 3}$

$n^{\log_2 3} \& n^{\log_5 3}$ — Here only constant difference but 'i' is in power, we cannot neglect it (eg: $n^3 > n^2$)

(3) Master Theorem (used to solve mostly divide & conquer Recurrence Rel^n)

master theo^m is valid for only this format questions

$\boxed{T(n) = aT(n/b) + f(n)}$ where $a \geq 1$, $b > 1$ and $f(n)$ is +ve f^n. Here a, b are constants.

eg:- given $T(n) = 8T(n/2) + n^5$

Compare with $T(n) = aT(n/b) + f(n)$

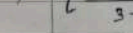
$\therefore a = 8,\ b = 2,\ f(n) = n^5$

Calculate $n^{\log_b a} \Rightarrow n^{\log_2 8} = n^3$ ⟹ compare with $f(n)$ (f(n) ⇓ n^5)

$\therefore n^3 < n^5$ ⟹ (whoever bigger that is answer) $\therefore O(n^5)$ Ans

* Suppose both are equal. i.e $n^{\log_b a} = f(n)$ (eg $n = n$)

then Answer is ⇒ $n^{\log_b a}(\log n)$ — take one person, multiply by log n



∵ tree contains log n levels.

CASE (i) If $f(n) = O(n^{\log_b a - \epsilon})$ → If we consider n^3 & n^5, n^5 is bigger by n^2 times

∴ Right side is bigger by n^ϵ times compared to left side.

where ϵ is constant, $\epsilon > 0$ | but n^ϵ is not constant i.e polynomial

∴ we can write $n^3 = n^{5-2} = \frac{n^5}{n^2}$

* comparing $f(n)$ & $n^{\log_b a}$, $n^{\log_b a}$ is bigger by n^ϵ times. where n^ϵ is a polynomial, ϵ is constant > 0

∴ Ans ⇒ $\boxed{T(n) = \Theta(n^{\log_b a})}$ because we proved $n^{\log_b a}$ is bigger than $f(n)$ by polynomial time

*Note:- Final Answer of Master Theorem gives Theta (Θ)
If case (i) fails, come to case (ii)

Case (ii):

if $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ → consider $n^5 + n^3$, $\therefore n^5 = \Omega(n^3)$

Here n^3 is smaller than n^5 by n^2 times $\therefore n^5 = n^{3+2}$

$\therefore$ comparing $f(n)$ & $n^{\log_b a}$, $n^{\log_b a}$ is smaller than $f(n)$ by n^ϵ times where n^ϵ is poly. & ϵ is constant > 0

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

$\because f(n)$ is bigger here, but * bigger by polynomial time

↑ this is imp here

Case (iii):- if $f(n) = \Theta\left(n^{\log_b a} \times (\log n)^k\right)$ where k is constant ≥ 0

↑ ∵ when both are equal, then for final answer we multiply by $\log n$.

Answer ⇒ $T(n) = \Theta\left(n^{\log_b a} \cdot (\log n)^{k+1}\right)$ ∴ when case (i) & (ii) fails, come to case (iii)

Que 1) $T(n) = 8T(n/2) + n^2$

Sol^n Acc. to master theorem:- $a=8$, $b=2$, $f(n)=n^2$

$f(n) = n^2$	$n^{\log_b a} = n^{\log_2 8}$
n^2	n^3

→ This is case (1) where right side bigger by n^1 times ($n^2 = n^3/n^1$) i.e polynomial times *

∴ Ans ⇒ $T(n) = \Theta(n^3)$

∵ n^3 is bigger

Que 2) $T(n) = 2T(n/2) + n^2$

Sol^n

$f(n) = n^2$	$n^{\log_b a} = n^{\log_2 2} = n^1$

⇒ This is case (2) where right side smaller by n times ($\because n^2 = n \times n$) i.e poly. time *

∴ Ans ⇒ $T(n) = \Theta(n^2)$ ∵ n^2 is bigger

Que 3) $T(n) = 2T(n/2) + n$

Sol^n

$f(n) = n$	$n^{\log_b a} = n^{\log_2 2} = n$

⇒ This is case (3) where both equal

* So, here we should ~~find~~ write right side interms of $\log n$ such that their equality doesn't affect

$f(n) = n$	$n^{\log_b a} = n \cdot (\log n)^0$

⇒ Now also they both are equal

∴ Ans ⇒ $T(n) = \Theta(n (\log n)^{0+1}) = \Theta(n \cdot \log n)$

Que 4) There is a recurrence rel^n $T(n) = 64T(n/2) + n^2$

Sol^n

$f(n) = n^2$	$n^{\log_b a} = n^{\log_2 64} = n^6$

⇒ Right side bigger by n^4 ($\because n^2 = n^{6-4}$) by polynomial time *

∴ Master theo. will work.

∴ Ans ⇒ $T(n) = \Theta(n^6)$

Que 5) $T(n) = 2T(n/2) + n \log n$

comparing with $aT(n/b) + f(n)$

$a=2, b=2, f(n) = n\log n$

$f(n)$	$n^{\log_b a}$
$= n\log n$	$= n$

⇒ Here, we can see that right side is smaller by $\log n$ times

Ans ⇒ $T(n) = \Theta(n\log n)$ ✗ ∴ This is wrong

Imp * BUT $\log n$ is logarithmic and not Polynomial

∴ case (ii) fail

∴ Go to case (iii) ⇒ Make left and Right equal by writing RHS in terms of $\log n$

$f(n)$	$n^{\log_b a}$
$= n\log n$	$n\cdot(\log n)^1$

⇒ Now both are equal asymptotically

∴ Ans ⇒ $T(n) = \Theta(n\cdot(\log n)^{1+1}) = \Theta(n\cdot(\log n)^2)$

* Note: This case (3) will be helpful when

a) $f(n)$ & $n^{\log_b a}$ are asymptotically equal

(or) b) $n^{\log_b a}$ is logarithmic times smaller than $f(n)$

Ques) consider a Recurrence rel^n where $f(n) = (\log n)^{20}$ and $n^{\log_a a} = 1$

Sol^n

$f(n)$	$n^{\log_b a}$
$=(\log n)^{20}$	$= 1$

⇒ Here RHS is smaller than LHS by $(\log n)^{20}$ times i.e logarithmic time

not polynomial time ∴ case 2 fail

∴ case 3 will work

Now, write RHS in terms of $\log n$ to make L.H.S = R.H.S.

$f(n)$	$n^{\log_b a}$
$(\log n)^{20}$	$1\cdot(\log n)^{20}$

⇒ Ans ⇒ $T(n) = \Theta(1\cdot(\log n)^{(20+1)})$

$T(n) = \Theta((\log n)^{21})$

∴ conclusion:- b/w LHS & RHS

- If there is no difference, then case 3
- If there is a Difference
  - case 3 → If logarithmic difference
  - case 1 (or) case 2 → If polynomial difference

Ques 7) $T(n) = 2T(n/2) + n^n$

Sol^n

$f(n)$	$n^{\log_b a}$
$= n^n$	$= n$

⇒ R.H.S is smaller than L.H.S by $\left(\frac{n^n}{n} = n^{n-1}\right)$ by n^{n-1} times

Note:- Here n^{n-1} is not polynomial, But more than polynomial

* So, Master Theorem works also for more than polynomial but not for less than polynomial.

∴ Ans ⇒ $T(n) = \Theta(n^n)$ ∵ n^n is bigger than

Que 1) $T(n) = 2T(\sqrt{n}) + \log n$

Sol^n

① Assume $n = 2^k$

$\therefore T(2^k) = 2T(2^{k/2}) + k$

② Assume $T(2^k) = S(k)$

$\therefore S(k) = 2S(k/2) + k$ — Now this is master theorem format.

③ Solving by Master theorem

$a = 2$, $b = 2$, $f(k) = k$.

$f(k)$	$k^{\log_b a}$
$= k$	$\Rightarrow k^{\log_2 2} = k$

Here L.H.S = R.H.S

$\therefore$ It is case(iii) of master theo

$\therefore$ write R.H.S in form of log. $\Rightarrow$ | k | $k(\log k)^0$ |

$\Rightarrow$ Ans = $\Theta(k \cdot \log k^{0+1}) = \Theta(k \cdot \log k) = S(k)$

Now, replace $S(k) = T(2^k)$ and $2^k = n$

$\Rightarrow k = \log_2 n$

$\Rightarrow T(2^k) = \Theta(k \cdot \log k)$

$\Rightarrow \boxed{T(n) = \Theta(\log_2 n \times \log(\log_2 n))}$

Que) $T(n) = T(\sqrt{n}) + c$ (constant)

Sol^n ① Assume $n = 2^k \Rightarrow k = \log_2 n$ | ② Assume $T(2^k) = S(k)$

$\therefore T(2^k) = T(2^{k/2}) + c$ | $\therefore S(k) = S(k/2) + c$

③ Solving by Master Theorem $f(k) = c$ | $k^{\log_b a} = k^{\log_1 1}$

$\Rightarrow k^0 = c$

Here L.H.S = R.H.S $\Rightarrow$ IIIrd case (iii)

$\therefore$ write RHS in terms of log $\Rightarrow$ | LHS: c | RHS: $c \cdot (\log k)^0$

$\therefore$ Ans $\Rightarrow S(k) = \Theta(\log k)^1$

④ Replace $S(k) = T(2^k)$ and $\because 2^k = n \Rightarrow k = \log_2 n$

$\therefore T(2^k) = \Theta(\log k)$

$\Rightarrow \boxed{T(n) = \Theta[\log(\log n)]}$

Que) $T(n) = 2^n T(n/2) + n^n$

Sol^n $\Rightarrow$ Here $f(n) = n^n$ & $n^{\log_b a} \Rightarrow n^{\log_2 2^n} = n^n$

Both are equal

But here $a = 2^n$ and $b = 2$

$\therefore$ a is not constant, Master theorem not Applicable

Go for Substitution method.

Que) $f(n) = n^2 \log n$ $\quad n^{\log_b a} = n$

Sol^n | $f(n) = n^2 \log n$ | $n^{\log_b a}$ |
|---|---|
| | n |

$\Rightarrow$ Here RHS smaller by $n \log n$ ($n^2 \log n = n \times n \log n$)

this is polynomial, not logarithmic

$\therefore$ Its case (ii)

$\therefore \boxed{\text{Ans} \Rightarrow T(n) = \Theta(n^2 \log n)}$ $\Leftarrow$

Ans

Note :- Non-recursive terms differ by only constant ∴ no change in answer

- $2T(n/2)+n = \theta(n\log n)$
- $2T(n/2)+2n = \theta(n\log n)$
- $2T(n/2)+20n = \theta(n\log n)$

BUT

- $2T(n/2)+\log(n) \neq \theta(n\log n)$ ← non recursive term differ by function and not constant

- $2T(n/2-10)+n$
- $2T(n/2+10)+n$

} comparing divide and substraction, Divide has more effect

∴ we can neglect Add^n & substraction

∴ Its equivalent to $2T(n/2)+n$ = ans → $\theta(n\log n)$

- $2T(n/2)+(-\log n)$ ← Not solved by master theorem ∵ $f(n)$ = +ve f^n.

Recursive terms differ constant

- $2T(n/2)+n = \theta(n\log n)$
- $T(n/2)+n = \theta(n)$

These two are not same. Actually these are 2 funct^n calls.

③ * Divide and Conquer

- It is one of the template (or)
- It is an algorithm designing technique

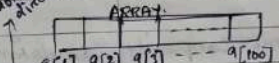
Divide the given big-problem into some sub-problems.

Conquer the sub-problems by calling recursively so that we will get sub-problem solution

Combine the subproblems solution so that we will get final problem

* Divide And Conquer Abstract Algorithm

we want to apply DAC algorithm on array 'a' which contains p to q elements where p is 1st element index and q is last index.



DAC(a, p, q)
{
if (Small(a,p,q)) - Terminat^n cond^n.
return (Solution(a,p,q))
else
{
m = Divide(a,p,q) - Divid
x = DAC(a, p, m) — Conqu
y = DAC(a, m+1, q)
z = Combine(x, y) — comb
return (z);
}
}

Small() { ... return(1); }
Solution() { ... }
Divide(a,p,q) { ... return(m); }
Combine(x,y) { ... x+(z) }

Problem to problem, these f^{ns} codes changes.

So, these fns are only templates, inside which code changes problem to problem (eg:- sometimes combine means 'multiplicat^n', sometimes addition, sometimes subtraction etc). Also there is no unique meaning for small ( ) f^n. Person-to-person its defn may change acc. to code written. And since small() is changing, automatically sol^n( ) is changing.

* Using this Divide & Conquer algo., we can solve many problems. such as quick sort, merge sort, binary search, finding max, min etc.

Divide & Conquer can solve many problems. For every other problem, there are many more algo. to solve them but the one which will take less time to solve is more effective and preferred. Eg:- There are many algo. which can solve quick sort. But solving quick sort by Divide and Conquer takes less time as compared to others.

Divide & Conquer

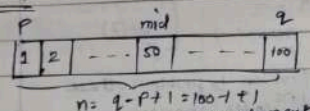
* Recursive Program for time complexity calculation
  1. Write Recurrence Relation for Time Complexity.
  2. Solve it using any method.

* Now, our array had 'n' elements.

Let us assume that to apply Divide & conq. on n elements array, Time complexity is T(n)

∴ DAC(n)

y = DAC(a, p, m)



n = q-p+1 = 100-1+1 = 100 elements

DAC(a, p, q) → No. of elements n = q-p+1 ∴ T.C = T(n)

if( ) → Terminat^n cond^n : Time comp = O(1)

Non recursive

else {

m = divide(a, p, q) → outside. Assume T.C = $f_1(n)$.

T.C depends on divide( ) f^n code which is written

Recursive call

x = DAC(a, p, m) → no. of elements = n/2 ∴ T.C = T(n/2)

y = DAC(a, m+1, q) → no. of elements = n/2 ∴ T.C = T(n/2)

Conquer time = 2T(n/2)

z = combine(x, y) → T.C depends on combine( ) f^n code

return(z)

let's assume T.C = $f_2(n)$

(it may be constant, O(n), O(n^2) etc. It depends on code written)

}

}

***

$$\therefore T(n) = \begin{cases} O(1) & \text{if } n \text{ is small} \\ f_1(n) + 2T(n/2) + f_2(n) & \text{if } n \text{ is big} \end{cases}$$

* Now,

Let f(n) be the time required for both divide and combine f^ns

where $f(n) = f_1(n) + f_2(n)$,

In $2T(n/2)$ ⇒ 2 is the no. of subproblems (no. of recursion calls), n/2 is the size of each subproblem

T(n/2) is Time comp. of one subproblem and T(n) is total Time comp. for the program.

$$\therefore T(n) = 2T(n/2) + f(n)$$

↓ In general

$$T(n) = aT(n/b) + f(n) \quad , a \geq 1, b > 1$$

**

Here a is no. of subproblems, n/b is size of each subproblem. T(n/b) is T.C of each subproblem. f(n) is cost of divide & combine.

* Applications of Divide And Conquer:
1. Finding Max & min
2. Power of an element
3. Binary search
4. Quick Sort
5. Merge sort.

(70% of algo. syllabus.)

6. counting no. of inversions
7. Continuous max^m subarray sum
8. Selection – procedure
9. Strassens matrix multiplication

Lecture-20

(1) Finding maximum and minimum.

I/P: An array of n-elements A[1:6] = [80, 20, 10, 200, 9, 78]

O/P: Find max & min^m : max = 200 ; min = 9

* First, without Divide & Conquer:-

Straight maxmin (a, n)

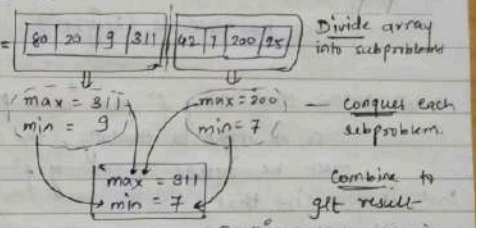
```
{
  max = a[1]
  min = a[1]
  for (i=2; i<=n; i++)        ⇒ No. of comparisions = n-1
                                 ∴ Time comp = O(n)
    if (max < a[i])
      max = a[i];
    else
      if (min > a[i])
      min = a[i];
  return (max, min);
}
```

⇒ This is a non recursive program where only one f^n call is there
∴ Stack space = 1

• worst case no. of comparisions = 2(n-1)
(1st fail (ie if max < a[i] fail) then go to 2nd comparisin (i.e if min > a[i]))

* using Divide and Conquer.

• Abstract:- Here what we will do is --

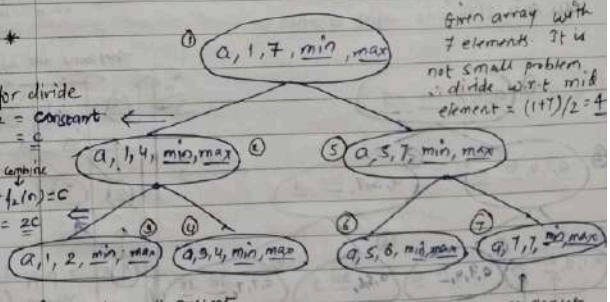


In this problem, meaning of combine is comparisin

• Now, our DAC problem contains small f^n ( ). ⇒ We can declare our problem is small when ⇒

if only 1 element present (eg: 10)	if 2 element present (eg: 10 > 20)
min = 10 max = 10	min = 10 max = 20
(0 comparisin)	(1 comparisin)

*

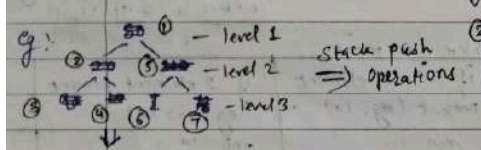


* Every circle is a f^n call ∴ Funct^n calling sequence : Preorder (1,2,3,4,5,6,7)
Funct^n execut^n sequence : Post-order (3,4,2,6,7,5,1)

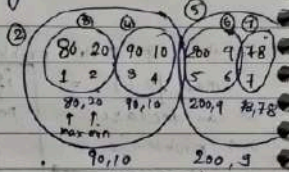
- Height of the tree = $\log_2 n$
  ∴ array contains 7 elements ⇒ $\lceil \log_2(7) \rceil = 3$
  ∴ Tree has 3 levels (height)

- We know that every f^n call is a push operat^n (stack)
  - So, we might be thinking, the stack space required must be equal to no. of f^n calls in the program
  but, it's not like that.
  Stack space depends on no. of levels in the tree. (k = stack space = no. of levels)

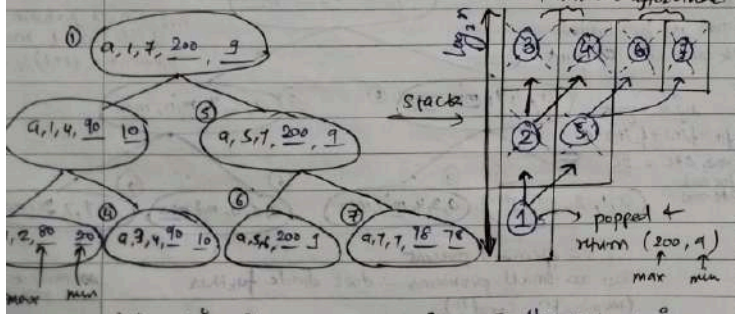
eg:



⇒ stack push operations



Every node at a particular level requires only one slot of stack. But nodes uses this slot one after another.



∴ Recursive program stack space = No. of levels in Recursive tree = K

we can get this value by solving recurrence relⁿ.

---

* Algo. to find max-min using divide & conquer

Consider an array of size n, with starting & ending element index is i and j respectively.

↳ ∴ size of array $n = j - i + 1$

T(n)

DAC maxmin (a, i, j)

① if (i == j)
  max = min = a[i];
  return (max, min);

→ small problem where one element present
eg: a, 7, 7, 78 (i, j; max = min) ⇒ 0 comparision

② if (i == j-1)
  if (a[i] < a[j])
  max = a[j], min = a[i];
  else
  max = a[i], min = a[j];
  return (max, min);

→ small prob where 2 elements present
eg: a, 5, 4, 90, 10 (i, j; max, min) ⇒ one comparision (a[i] < a[j])

Here, ① and ② are small (-) and solution ( ) function codes written (filling the gaps) for 1 & 2 elements

Small problem T.C = constant = O(1)

③ else
  mid = ⌈(i+j)/2⌉ ⇒ Divide
  ↳ no loop ∴ T.C = constant = O(1)

T(n/2) ← (max1, min1) = DAC maxmin (a, i, mid); ⇒ Recursive code
T(n/2) ← (max2, min2) = DAC maxmin (a, mid+1, j); ⇒ Conquer
↳ T(n/2) + T(n/2) = 2

  if (max1 > max2)
  max = max1 else max = max2
  if (min1 < min2)
  min = min1 else min = min2
  return (max, min)

⇒ combine
↳ no loop ∴ T.C = O(1)

Let $T(n)$ be the Time complexity of above program.

Recurrence Relation : $T(n) = \begin{cases} O(1) & \text{if } n=1 \text{ or } n=2 \\ O(1) + 2T(n/2) + O(1) & \text{if } n>2 \end{cases}$

$\therefore T(n) = 2T(n/2) + \underbrace{c}_{\text{Constant}}$. Here 'c' is the amount of time req. for both divide and combine at 1st level only

Now, solve by Substitution

$T(n) = 2\left[2T(n/2^2) + c\right] + c$

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

(2c : 2nd level, c : 1st level divide + combine time)

$T(n) = 2^3 T(n/2^3) + 2^2 c + 2^1 c + c$

(3rd level, 2nd level, 1st level, Divide & combine time)

minimum cond^n. $T(1) = 1$

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

$= 2^{k+1}$

$= \log_2 n - 1$

stack space

K times

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

$= \frac{n}{2} T(2) + c\left[2^0 + 2^1 + 2^2 + \cdots - 2^{\log_2 n - 2}\right]$

last level ⇒ small problem. Big problem. GP with $r > 1$

$= \frac{n}{2} \cdot O(1) + c\left[\frac{1\left(2^{\log n - 1} - 1\right)}{2-1}\right]$

$= \frac{n}{2} + c \cdot \frac{2^{\log n}}{2} \Rightarrow \frac{n}{2} + c \cdot \frac{n}{2} = O(n)$

Note :- Finding max, min in n-element array for any method.

Time comp. = $O(n)$ → best, worst, avg case. ⇒ because for else part of the code, we have to run entire code, we have no cond^n of stopping in between. (∵ return is at end of the code)

Because any how we must scan each and every element of the array. (1 entire scan needed)

Let $C[n]$ be the no. of comparisions b/w the elements on n-elements array of above program.

DAC maxmin ( ) ⇒ $C[n]$

{
① { 0 comparisions }

② { 1 comparision if(a[i] < a[j]) }

③ {
divide code ⇒ 0 comparisions
conquer code ⇒ $C[n/2]$, $C[n/2]$
combine 2 comparisions code ⇒ if(max, >max2) if(min1<min2)
}
}

⇒ $C(n) = \frac{n}{2} \cdot C(2) + \left[\frac{2\left(2^{\log n - 1} - 1\right)}{2-1}\right]$

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

∴ Recurrence Relation.

$C[n] = \begin{cases} 0 & \text{if } n=1 \\ 1 & \text{if } n=2 \\ 0 + 2C[n/2] + 2 & \text{if } n>2 \end{cases}$

$C(n) = 2C[n/2] + 2$ ← no of comparisons needed for both divide & combine at 1st level

Now, solve by substitution

$C(n) = 2^2 C(n/2^2) + 2^2 + 2^1$

$C(n) = 2^3 C(n/2^3) + 2^3 + 2^2 + 2^1$

$k \Leftarrow \log n - 1$ times

$C(n) = 2^{\log n - 1} C\left(\frac{n}{2^{\log n - 1}}\right) + \left[2^1 + 2^2 + \cdots + 2^{\log n - 1}\right]$

$\therefore C(n) = \frac{3n}{2} - 2$
$= 1.5n - 2$

every case (Best, worst, avg)

Here we cannot ignore any constants because we are calculating no. of comparisions not time complexity

Note :- In the Recurrence rel^n of no of comparisions, we cannot apply master Theorem. Because MT gives approximate answer. But to calculate no. of comparisions we must find exact answer

- Here Stack space = $k = \log_2 n - 1$

Que) i/p: An array of n-distinct integers
o/p: Find max & min.
Time complexity ?

Sol^n we have to do complete scan (1 loop compulsory) which is also equal to comparisions

$\therefore$ Time complexity = n = $O(n)$ = $\Omega(n)$ = $\Theta(n)$

Options: a) $O(n)$ b) $O(n^4)$ c) $O(n^2)$ d) $O(n)$ e) $\Omega(n)$ f) $\Omega(1)$ g) $\Omega(n^3)$

Que) Given (Sorted) array · Finding max, min element
Time complexity?

Sol^n $O(1)$ $\Rightarrow$ array : 2, 8, 9, 14, 70
↑ min ↑ max
return (a[1], a[n])

Que) An array of n-distinct integers.
Find any element which is neither max nor min.

Sol^n array :- 20 50 10 60 5 80 23

Take first 3 elements $\Rightarrow$ Sort them : 10, 20, 50
Then Here mid element 20 is the answer.
20 is not maximum $\because$ 50 > 20
Also 20 is not minimum $\because$ 10 < 20

Time complexity for sorting 3 elements = constant = $\Theta(1)$ (every case)

Que) An array of n distinct elements. Find any element which is neither 2nd max nor 2nd min.

Sol^n array: 20, 50, 10, 60, 5, 80, 23, 29

Sort the 5 elements

5, 10, 20, 50, 60

This is the answer.
$\because$ 20 is greater than 2 elements and less than 2 elements.

(2) Power of an element

i/p: Two integers $a \geq 2$, $n \geq 1$
o/p: Find a^n

$\Rightarrow$ $2^5 = \underbrace{2 \times 2 \times 2 \times 2 \times 2}_{5 \text{ times}} = 32$

without pow

```
s=1
for(i=1;i<=n;i++)
    s=s*a
return(s)
```

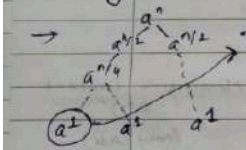
$O(n)$

• With Divide and Conquer.

→ Here Divide ⇒ a^n → $a^{n/2}$. $a^{n/2}$

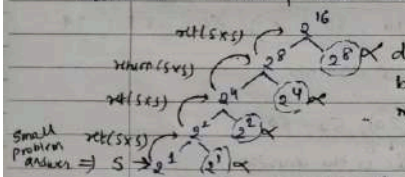
→ Here combine ⇒ means multiplication

→ ∵ Here also, we are dividing by 2 ; ∴ No. of levels $= \log_2 n$

→ 

This is small prop problem
if n == 1 , then return (a) ⇒ O(1)

→ Here, we will expand only left child ∴ It will result in a unary tree rather than binary tree.



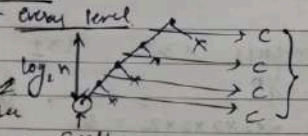
don't expand right child but rather simply return multiplication value from single left child only

∵ Divide time = constant, Combine time = constant. } c

at every level ⇒ total cost = constant

Here no series will come ∵ Its unary tree ant at every level

only 1 fn call.



$c \cdot \log_2 n$

height of tree = Stack space 'k'

Small problem → O(1)

total = $O(1) + c \cdot \log n = O(\log_2 n)$

Note :- If we do this problem by previous approach i.e expand both left and right subtree. we will get same answer but time complexity will be more ∵ It will lead to a series ⇒ $c + 2c + 2^2c + 2^3c + \cdots + 2^{\log n}c$

⇒ Time comp here $= O(n)$

→ Algorithm to find Power of an element using DAC.

DAC-Power (a, n) ⇒ T(n)

```
if (n==1) return(a)  ⇒ Small problem. = O(1)
else
    mid = ⌊n/2⌋        ⇒ Divide = Constant time = c1
    S = DAC-Power (a, mid)   ⇒ Conquer ⇒ T(n/2)
    C = S×S            ⇒ combine = constant time = c2
    return (C);
```

Let T(n) be the Time complexity of above problem.

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ O(1) + T(n/2) + O(1) & \text{if } n > 1 \end{cases}$$

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

Solve this by substitution

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

$T(n) = T(n/2^3) + c + c + c$

↓ log n times

$T(n) = T\left(\frac{n}{2^{\log n}}\right) + c \cdot \log n$

$T(n) = T(1) + c \cdot \log n$.

$\Rightarrow T(n) = O(1) + c \cdot \log n$

(terminating cond^n) → cost for small problem; cost for big problem.

$\therefore$ **Time complexity = $O(\log n)$** ← Best, worst, Avg case.

∵ In else part of the code, we cannot stop in between. we can stop at ending only ∵ return statement is at end of code.

→ **Stack space = $k = \log n$**

- Let $T(n)$ = No. of multiplications in the above program.

```
DAC-power(a,n)
if ( 0 multiplicatn
     operatns )
else {
  Divide code ⇒ 0 multiplicatn operators.
  conquer code ⇒ T(n/2)
  combine code ⇒ 1 * operator
}
```

$$\Rightarrow T(n) = \begin{cases} 0 & \text{if } n=1 \\ 0 + T(n/2) + 1 & \text{if } n>1 \end{cases}$$

$\Downarrow$

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

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

$= T(n/2^3) + 1 + 1 + 1$

$\downarrow \log n$ times

$T(n) = T(n/2^{\log n}) + \underbrace{1 + 1 \cdots 1}_{\log n \text{ times}}$

$= T(1) + 1 \cdot \log n$

$= 0 + \log n$

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

* Searching :- Linear Search

i/p: An array of n-distinct elements, and element-x.

o/p: Find position of element x in the array.

eg:- A[1..10] = [50, 70, 60, 19, 29, 89, 11, 21, 81, 60]

- In the array 8th index value ⇒ a[8] = 21 ← random access possible
- In the array what is the index of value 21 ? ← we have to scan the array. Random access not possible.

i/p (value)	x=21	x=60	x=19	x=500
o/p (index)	8	10	4	-1

→ • If array contains 500 elements / If array contains 7lac elements } Best case time complexity to find an element = $O(1)$

∵ The element might be present at 1st position only.

∴ Best case time complexity never doesn't depend on no. of elements in the array. But it depends on logic of code written here

→ But worst case T.C = $O(n)$ ← element present at last index

Note :- • Linear search will give Best case T.C when, whatever element we are finding, is present at first or surrounding to first element

∵ 1 element matching = 2nd element matching = 3rd — — — Time complexity of all = Constant = $O(1)$

• — 11 — worst case — 11 — present at last or surrounding to last element = $O(n)$

A = [50, 70, 60, 19, 29, 39 - - - -600, 42, 480]

index → 1 2 3 4 5 6 - - - n-3 n-2 n-1

$O(1)$ $O(n)$

• Average case Time complexity of linear search.

- To find avg case, we have to take care of all time complexity for all comparisions

$$\therefore \text{Avg case} = \frac{1 + 2 + 3 + 4 + 5 \quad \cdots \quad (n-1) + n}{n \text{ elements}}$$

(1 ← best, n ← worst, numerator ← no. of comparisions)

$$\text{Avg case} = \frac{n(n+1)/2}{n} = \frac{n+1}{2} = O(n)$$

∴ Asymptotically, worst and avg case of linear search are same, but mathematically both are different.

* Algo for linear search.

```
for (i=1; i<=n; i++)
{
    if (a[i] == x)
        return (i)
}
If (unsuccessful)
    return (-1);
```

(3) Binary Search.

I/P: Sorted array of n-elements and element-x

O/P: If element-x is found, return position of x otherwise return -1.

*Note: If we are given an array, which searching is better. Linear or Binary search?

→ It depends on whether array is sorted or unsorted.

Unsorted ⇒ Apply to linear search better ⇒ $O(n)$

Sorted ⇒ Binary search better = $O(\log n)$

• If array is unsorted → And if we try to sort the array and then apply Binary search then

Sorting + Binary search

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

∴ Here for unsorted array ⇒ linear search better ⇒ $O(n)$

Eg:-

	1	2	3	4	5	6	7
A	50	60	80	100	200	300	400

If element x = 100
Calculate mid = $\frac{1+7}{2} = (4)$
a[4] = 100
return mid

if element-x = 60
Calculate mid = $(1+7)/2 = 4$
∵ already a[4] = 100 & x = 60 < a[4] = 100
Binary search ⇒ index 1 to 3.
mid = $\frac{1+3}{2} = 2$

if element x = 300
mid = $\frac{1+7}{2} = 4$
∵ a[4] = 100 < x = 300
B.S ⇒ index 5 to 7
mid = $\frac{5+7}{2} = 6$

• Here, In index if only 1 element present, then it is small problem (eg: index 2 to 2)

- If we find our element in 1st division only, (a[mid] = x) then its Best case.
- If we find our element in nth division i.e we divide the array by half every time until only one element present then its worst case.

* Algorithm for Binary search

we will apply Binary search in given array a with first index = i, last index = j, element to search = x

⇒ BS (a, i, j, x) = T(n)

```
if (i == j)        ← small problem
                     ∵ 1 ele present = constant time = c
  if (a[i] == x)     = O(1)
    return (i);
  else return (-1);
else                 ← i is index
  mid = ⌊(i+j)/2⌋    } ⇒ DIVIDE
  if (a[mid] == x)     constant time (no loop)
    return (mid);      = c ⇒ this here Best case
  if (x < a[mid])
    BS (a, i, mid-1, x);   ⇒ T(n/2)   } Two 1st calls but only one will execute depending on the condn.
  else
    BS (a, mid+1, j, x);   ⇒ T(n/2)
```

CONQUER

∴ This program is tail recursion. Here combine not required. ∵ NO work after recursive call

- Let T(n) be the time complexity of the above program

→ Best case T(n) = O(1) ← ∵ In else part of the code, we can stop in between ∵ return statement is not at last but is there in between.

Here stack space = O(1) (space comp.)

Best case.

∴ we will have separate Best, worst & avg case complexities

→ Recurrence Rel^n for worst case

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ c + T(n/2) & \text{if } n > 1 \end{cases}$$

⇓

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

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

$= T(n/2^3) + 3c$

↓ logn time

$= T(n/2^{\log n}) + c \cdot \log n$

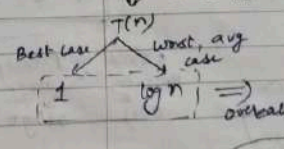
$T(n) = T(1) + c \cdot \log n = O(1) + c \cdot \log n$

⇒ $\boxed{T(n) = O(\log n)}$

worst case: stack space = logn

→ Average case ⇒ It is also close to worst case = O(logn)

* Time comp of Binary Search



overall

$= \Omega(n)$ ✗
$= O(\log n)$ ✓
$= \Omega(1)$ ✓
$= O(\log n)$ ✗
$= O(n)$ ✓

$(\because 1 = O(\log n)$ but $1 \neq \Omega(\log n))$

* Binary search worst case Time complexity

T(n) ⇒ logn

$= O(\log n)$ ✓
$= O(n)$ ✓
$= O(n^3)$ ✓
$= \Omega(1)$ ✓
$= \Omega(n)$ ✗

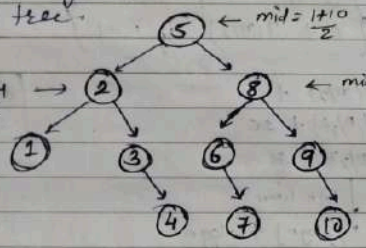
∴

GATE
Que) Binary search, No. of elements = n . Avg. no of comparision ?

Soln

- If n is very big no. then
  Avg comparisions = $\log_2 n$
- If n is small no.
  eg : 10 elements (1, 2, 3, - - - 10)

Then avg. no of comparisions is calculated by making a binary tree.



Total comparisions = Total comparisions for each element at every level

$= 3 \times 4 + 4 \times 3 + 2 \times 2 + 1 \times 1 = 29$

(3 ele. at last level; 4 comparision for each element)

$\Rightarrow$ Avg $= \frac{29}{10} = 2.9$

* Non recursive Prog. for Binary search

```
BS(a, i, j, x)
while(i <= j)
{
    if(i == j)
    {
        if a[i] == x
            return(i)
        else return(-1)
    }
    else
    {
        mid = (i+j)/2
        if(a[mid] == x)   return(mid);
        else if(x < a[mid])   j = mid - 1;
        else   i = mid + 1;
    }
}
```

- Time complexity
  Best case: $O(1)$; worst case: $O(\log n)$
- Stack space = 1 (every case) (∵ recursion may there)

* Note :- It is not that everytime non recursive prog will take less stack space.
Non recursive program also may take more stack space sometimes in some other programs
But most of the time non recursion takes less stack space

(ii) For Tail Recursive Program, writing non recursive program is very easy $\Rightarrow$ Just keep the code which is before recursive call in a while loop.

Que) i/p: A sorted array of n - distinct elements.
o/p: Find any 2 elements (a, b) such that
① a + b = 500
② a + b < 500
③ a + b > 500

	1	2	3	4	5	6	7
a	100	200	300	400	500	600	700

Soln

② For finding $a + b < 500$ $\Rightarrow$ return (a[1], a[2]) $\Rightarrow O(1)$
since array is sorted
then first 2 element sum must be less than 500, if its not less than 500 then there are no other elements in that array whose sum is less than 500

③ Similarly for $a + b > 500$ $\Rightarrow$ return (a[n-1], a[n]) $\Rightarrow O(1)$
→ return last 2 elements.

① Now for $a + b = 500$ $\Rightarrow$ There may be many cases (eg : 100+400, 200+300)

Apply linear search for each element.

(M1)

```
for (i=1 ; i≤n ; i++)
    for (j=i+1 ; j≤n ; j++)
        if (a[i] + a[j] == 500)
            return (a[i], a[j])
```

A [100 200 300 400 500 600 700]

⇒ 'n' times linear search on 'n-1' pairs

⇒ n × n = $O(n^2)$

(M2) Apply Binary search

A [100 (200 300 400) 500 600 700]

mid = j

fix i ∴ i+j = 600 which is > 500 ∴ go left and again apply Binary search ⇒ mid = 300

i+j = 100+300 = 400 < 500 ∴ go right

∴ n times Binary search on n elements

= n × log n = $O(n \log n)$

Greedy Algo

(M3) Find any 2 ele. such that a+b = 1300

A [100 200 300 400 500 600 700]

a b

a+b = 500 which is < 1300

∴ Increase a (shift right)

& check every time.

worst case a reaches its end where a=600, b=700, a+b = 1300 ⇒ $O(n)$

such that a+b = 300

A [100 200 300 400 500 600 700]

a b

a+b = 800 which is > 300

∴ left shift b and compare every time.

worst case b=200, a=100

a+b = 300

= $O(n)$: one entire scan

8) Given unsorted array of n-distinct elements.

find any 2 elements a, b such that

(i) a+b < 500

If array is not sorted

Method (1)

- First sort the array ⇒ $O(n \log n)$
- return 1st, 2nd elements ⇒ $O(1)$

$O(n \log n)$

Method (2)

- Find 1st minimum element ⇒ $O(n)$
- Find 2nd minimum element ⇒ $O(n)$

$2n = O(n)$

(ii) a+b = 500

If array is not sorted, (no need of sorting here)

(M.1) : linear search ⇒ $O(n^2)$

(M.2) : Binary search ⇒ (same as on prev. page) ⇒ · First sort = $n \log n$ · Apply BS = $n \log n$

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

(M.3) : Greedy Algo ⇒

- First sort = $n \log n$
- Apply Greedy algo = n

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

(4) Merge Sort :

I/P : An array of n-elements

O/P : Sorted array of n-elements.

*Note :- Merge sort will give time complexity as $n \log n$ whether we give it sorted or unsorted array

BUT

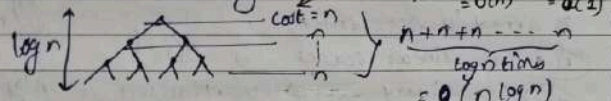
In case of Quick sort, If we give it unsorted array it will give $Tc = n \log n$ but if we unfortunately give it sorted array, it will punish you and give $Tc = n^2$

* In Merge sort: we divide array continuously Blindly in 2 parts (with resp. to mid)

Therefore Dividing take $O(1)$ time.

But here major role is played by Combine which will take $O(n)$ time for each level.

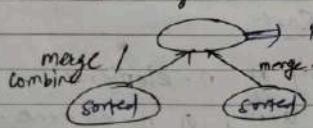
No. of such levels are $\log n$



* Here Major Contribution is of Combine

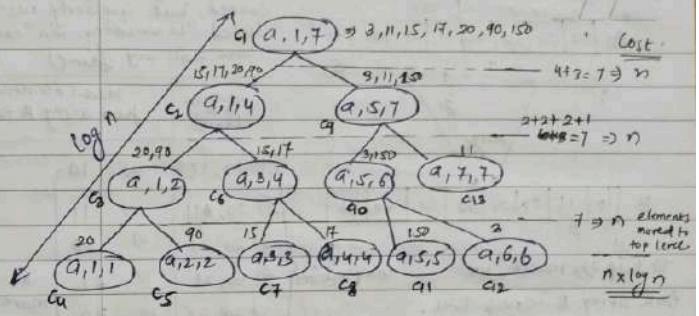
But in Quick Sort, major contribution is of Divide

* Note: Merge sort is nothing but merging 2 sorted subarrays.



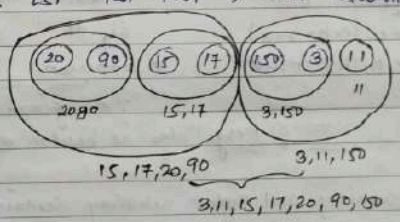
* 1 element only present :- is a small problem (already sorted)

Eg:- A [20 | 90 | 50,15 | 17 | 150 | 3 | 11]
1 2 3 4 5 6 7



Preorder : Root LST RST → Funct^n calling order

Postorder: LST RST Root → Funct^n execution order.



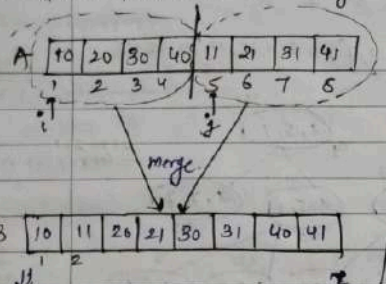
* Merge Algorithm.

Note that It's merge algo, and not mergeSort

algorithm :→ (Papa)

For MergeSort, Input is 'An array of n elements' but for Merge Algo. Input is 'Two sorted subarrays' to which we will merge to form one single sorted array of elements.

Eg:- Given two sorted subarrays. (merge Algorithm)



To keep the result, we took array B in next level. That why mergesort is a best example for outplace algorithm

An outside space (extra space) we are taking to store result.

These 2 array are individually sorted, but combinedly not sorted. (on combining, min^m can come from any group)

Compare (i, j)		Moved element from array A to B
1) 10, 11	⇒	10
2) 20, 11	⇒	11
3) 20, 21	⇒	20
4) 30, 21	⇒	21
5) 30, 31	⇒	30
6) 40, 31	⇒	31
7) 40, 41	⇒	40
8) 41	⇒	41

8 more

no comparision — but more needed

∵ no. of more oprⁿˢ is more.

∴ T.C of merge algo. is based on move oprⁿˢ only.

* If left subarray and Right subarray contains m, n elements respectively. then

No. of comparisions	No. of move operations
m+n-1	m+n

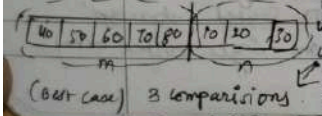
worst case

Time complexity = more operⁿˢ. (∵ its greater)

= O(m+n)

every case

* For Best case no of comparisions = min(m,n)



(Best case) 3 comparisions

Imp

** Note :- 1) we cannot apply merging if two subarrays are not sorted. we should first sort them individually to apply merge algorithm

** 2) Merging 2-sorted subarrays each of size m and n will take Time complexity = O(m+n) (every case)

- It is a outplace merge algorithm.

* In Divide & Conquer : we are dividing arrays into half every time, m = n = n/2

∴ Time comp. for merge = O(n/2 + n/2)

(or moves) — operation = O(n) (every case)

(not merge sort algo)

* Merge-Sort Algorithm

mergesort (a, i, j) ⇒ T(n)

```
if(i==j) return (a[i]);
else
  mid = ⌊(i+j)/2⌋   → Blindly divide array into 2 parts. ⇒ constant time
  T(n/2) mergesort(a, i, mid);   } These f^n calls will sort left & right
  T(n/2) mergesort(a, mid+1, j); } subarrays
  merge(a, i, mid, mid+1, j)
  return (a);
```

(1st half n/2 + 2nd half n/2 ⇒ outplace ⇒ n space extra)

(copy all elements from array b to a and then return a.)

a: sorted subarrays

Let $T(n)$ be the Time complexity of above algorithm

Then Recurrence Relⁿ

$$T(n) = \begin{cases} O(1) & \text{if } n=1 \\ c + 2T(n/2) + n & \text{if } n>1 \end{cases}$$

$\Rightarrow T(n) = 2T(n/2) + n$ ← Divide time & combine time at 1st level. But we have $\log n$ levels.

Conquer

∴ Solve by substitution

$\Rightarrow T(n) = 2\left[2T\left(n/2^2\right) + n/2\right] + n$

$= 2^2 T\left(n/2^2\right) + n + n$ ← 1st level & 2nd level non recursive term same

$T(n) = 2^3 T\left(n/2^3\right) + n + n + n$ (3rd level, 2nd level, 1st level) divide + combine time.

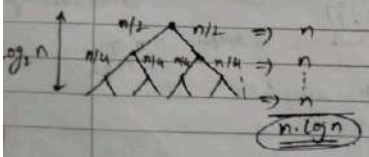
$\log n$ times

$T(n) = 2^{\log_2 n} T(1) + n\log n$ ← Divide + combine time at all levels.

Divide + combine time at last level (small problem)

$\Rightarrow T(n) = n + n\cdot\log n$ $\Rightarrow$ $\boxed{T(n) = O(n\log n)}$ (every case) (outplace)

∴ In else part return statement is at least. we cannot stop in between



- Space complexity for merge sort:
  - 1st is given by extra space taken : one is taken by usual stack i.e $\log n$ (stack space is equal to height of tree and not no. of fn calls)
  - 2nd is taken by merge algo (outplace)

∴ Space comp = Extra space (stack space $= \log n$; merge algo $= n$ (outplace)) $\} O(n)$

Qᵘᵉ) i/p : $\log n$ sorted subarrays each of size $\frac{n}{\log n}$

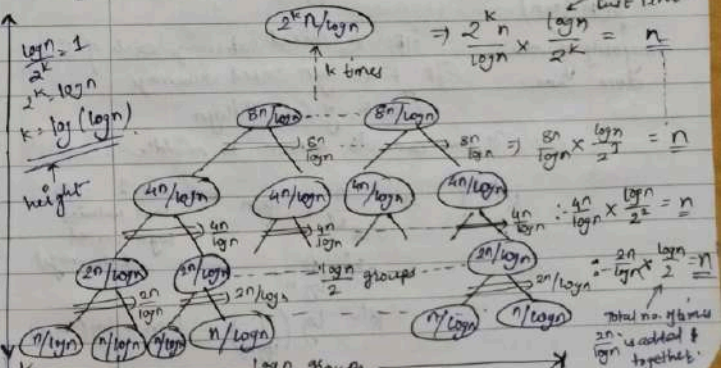
o/p: Find single sorted array with all elements

Find time complexity.

Solⁿ Given $\log n$ number of sorted subarrays in last row:

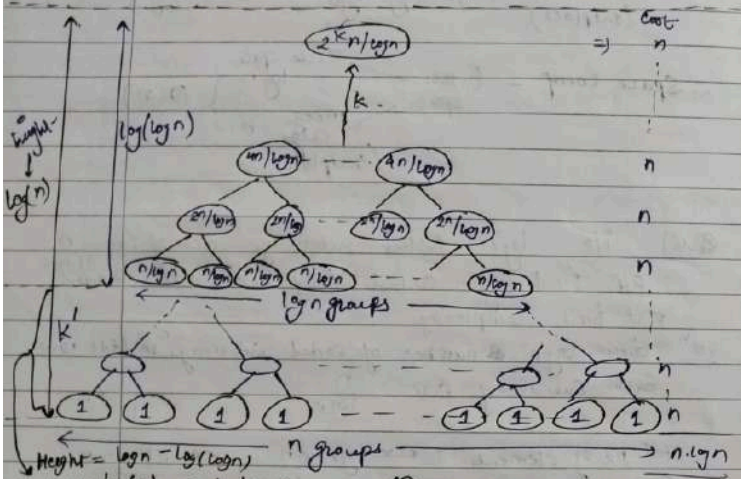
Each subarray size $= \frac{n}{\log n}$.

∴ Total no of elements in single sorted array $= \log n \times \frac{n}{\log n} = n$ elements



∴ Time complexity = $n+n+n+\cdots+n$ ($\log(\log n)$ times) $= n\cdot\log(\log n)$

$\boxed{T.C = O(n\cdot\log(\log n))}$



i/p: n sorted subarrays each of size $\frac{n}{n}$

o/p: Find $\log n$ sorted subarrays each of size $n/\log n$

In this case do bottom to middle.

$k' = ? \Rightarrow \frac{n}{2^{k'}} = \log n$ ← (not 1 ∵ last level containing $\log n$ sorted subarrays)

$2^{k'} = \frac{n}{\log n}$

$k' = \log\left(\frac{n}{\log n}\right) = \log n - \log(\log n)$

Que) i/p: $\frac{n}{a}$ sorted subarrays each of size $\frac{n}{b}$

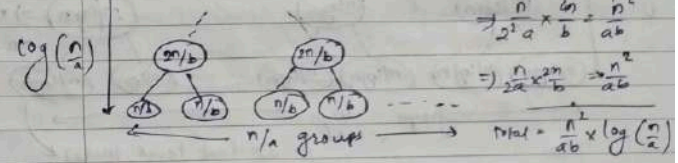
o/p: Single sorted array with all elements

sol^n groups at last level = n/a

groups at level above last level = $n/2a$

groups at above this level = $n/2^2a$ .....

∴ No. of levels = $\log\left(\frac{n}{a}\right)$



* Note:
- If give More than one sorted subarray
  ⇒ (a) Apply Merge Algorithm (It may work) (or) try
- only one sorted subarray.
  ⇒ Apply Binary search (It may works) (or) try

Que) i/p: $\log n$ subarrays each of size $n/\log n$ where every subarray is sorted using Bubble sort.

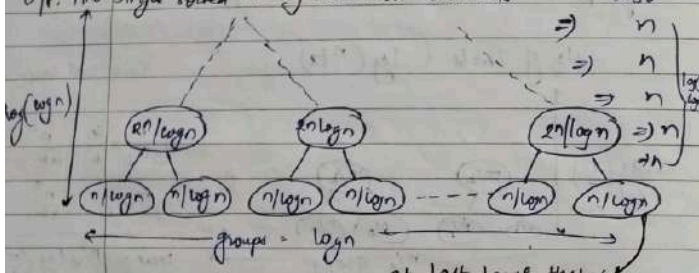
↑ these subarrays are sorted. ∴ we can apply merge sort but its told to sort using Bubble sort ($T.C = n^2$)

↑ where, subarray not sorted ∴ cannot apply merge sort

In this question, sub arrays are not sorted. But if we want to apply merge algo, subarrays should be sorted first.

This sorting is done by Bubble sort.

Q/p: Find single sorted array with all elements.



at last level there is no merge oper^n ∵ these subarrays are small problem and are sorted using Bubble sort ($T_C = n^2$)

∴ Time complexity / cost at last level $= \log n \times \left(\frac{n}{\log n}\right)^2 = \frac{n^2}{\log n}$

(no. of groups; T_C for Bubble sort)

Overall, Time complexity for all levels $= O\left[n \cdot \log(\log n) + \frac{n^2}{\log n}\right]$

Before this question, last level was already sorted but here last level sorting cost also added.

*Note:- Merge sort is better for larger arrays (Big Problem) BUT for small arrays (Small problem) (Last level generally) INSERTION SORT (or) BUBBLE SORT is always better

* preferred when array is almost sorted, very less sorting required. Here quick sort is not preferred.

* fibonacci merge sort.

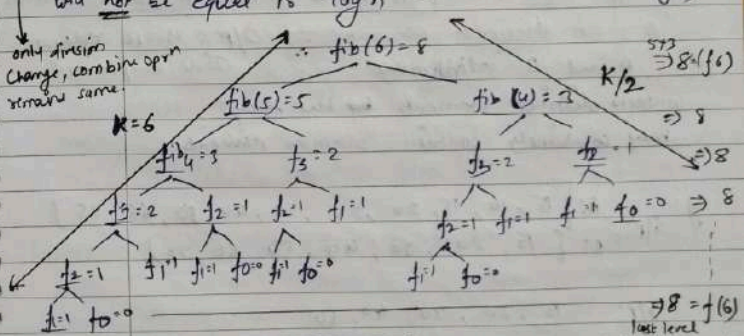
n	0	1	2	3	4	5	6	7	8	9	10
fib(n)	0	1	1	2	3	5	8	13	21	34	55

we know that fib(6) = 8. If we consider normal merge sort, we would divide fib(6) = 8 → 4, 4

But in fibonacci merge sort, we are dividing fib(6) = 8 ⇒ fib(5) = 5, fib(4) = 3 (5, 3) → unequal division.

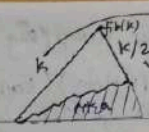
So, here we are not dividing always by 2. ∴ No. of levels (height) will not be equal to log n

only division change, combine oprn remains same.



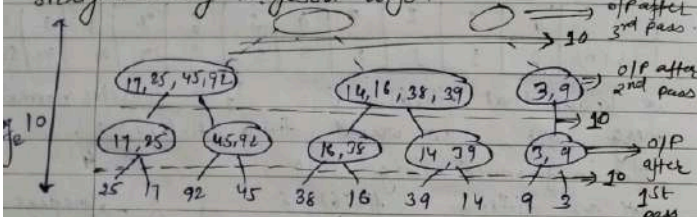
$\therefore$ Time comp $= O[6 \times fib(8)]$

$= O[k \cdot fib(k)]$



$O[k \cdot fib(k)]$, $\Omega[\frac{k}{2} \cdot fib(k)]$ } $O[k \cdot fib(k)]$ (∵ only constant difference)

Que) Consider array of 10-ele (25, 17, 92, 45, 38, 16, 39, 14, 9, 3). What is the O/P after 2nd pass of straight 2-way merge sort algo.



Straight 2 way merge sort

Que) I/P: 2 sorted array A and B. A ⇒ m distinct elements, B ⇒ n distinct elements ⇒ O/P: A∩B elements. Time complexity?

A and B individually contain distinct elements but they may commonly contain common elements.

eg: I/P: A: [5, 10, 15, 20, 25, 30, 40, 50, 65, 75]
B: [10, 20, 30, 40, 50, 60, 70]

O/P: 10, 20, 30, 40, 50

(M1): m × linear search(n) = m × n = O(m·n) $\xrightarrow{\text{if } m=n}$ $O(n^2)$ (W.C)

(M2): m × Binary search(n) = m × log(n) = O(m·log n) $\xrightarrow{m=n}$ O(n log n) (worst case)

(M3): Modified Merge ⇒ O(m+n) $\xrightarrow{\text{if } m=n}$ O(n) (worst case)

A[5, 10, 15, 20, 25, 30, 40, 50, 65, 75] (i)

B[10, 20, 30, 40, 50, 60, 70] (j)

only one time scan of A & B = O(m+n)

- compare i & j ⇒ ∵ i < j ⇒ i = i+1
- compare i & j ⇒ ∵ i = j ⇒ print(i, j) ⇒ i++, j++.
- Repeat this steps --- till i = 65, j = 60
- Next compare (i, j) ⇒ i > j ⇒ j = j+1 — Repeat these steps.

(5) Quick Sort

1. It is Divide & Conquer application
2. It is Inplace (unlike merge sort which is outplace)
3. It is not stable. (It may fail)

(Merge sort is stable) i.e Repeatative element order doesn't change after sorting

eg: I/P: 10_a 20 10_b 40 10_c 25

After Merge sort O/P: 10_a 10_b 10_c 20 25 40

But Quick sort may fail (orders may change)

4. It is practically used sorting algorithm
   - Because Quick sort follows meaningful division unlike merge sort which divides blindly from middle.