NAME: Rakesh Nama STD.: ______ SEC.: ______ ROLL NO.: ______ SUB.: Algorithms

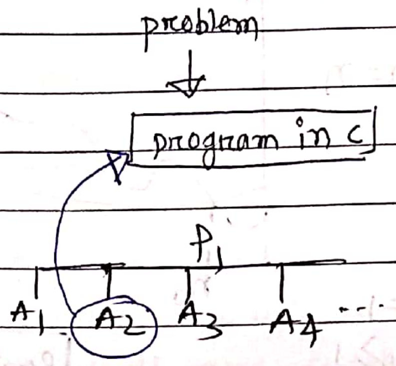
S. No.	Date	Title	Page No.	Teacher's Sign / Remarks
1.		Time and Space Analysis.	1	
2.		Sorting Techniques.	2	
3.		Greedy Algorithms.	3	
4.		Dijkstra Algorithms.	4	
5.		Bellman-ford algorithm.	4	
6.		Shortest paths in DAGs.	4	
7.		Dynamic programming. + Matrix Chain - Multiplication.	5	
8.		Longest common subsequence.	6	
9.		(Multi stage graph. ~~Eka~~ +	6	
10.		knapsack).		
11.		~~Travelling Salesman problem~~. Subset sum	6	
12.		Travelling Salesman problem.	7	
13.		All pairs shortest path - Floyd Warshall.	8	
14.		NP Completeness. (Not req for gate)	8	
15.		+ problems on NP completeness.		

Scanned by CamScanner

Time and space analysis

www.gatenotes.in

- Introduction to asymptotic notations =



analysis of an algorithm by — (I) Time (less time)
(II) Memory (less memory)

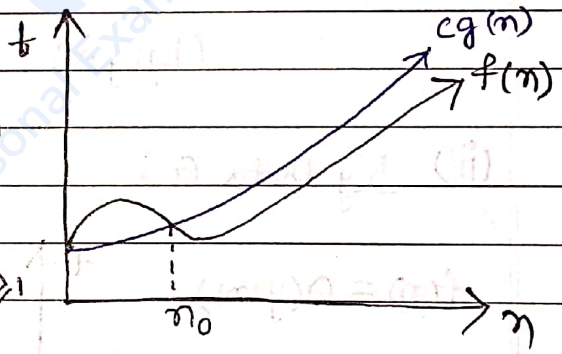
(i) Big (oh) O :

ex:

$f(n) = 3n+2,\ g(n) = n$

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

$f(n) \leq c\,g(n),\ c > 0,\ n_0 \geq 1$



$3n+2 \leq cn$.

$c = 4$

$3n+2 \leq 4n$

$n \geq 2$

$$\boxed{\begin{array}{l} f(n) \leq c\,g(n) \\ n \geq n_0 \\ c > 0,\ n_0 \geq 1 \end{array}}$$

$$\boxed{f(n) = O\,g(n)}$$

So, $f(n) = 3n+2$; $g(n) = n$

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

$g(n) = n$ → always go for least upper bound.
$= n^3$
$= n^4$ here least upper bound is 'n'.
$\vdots$
n^n
2^n

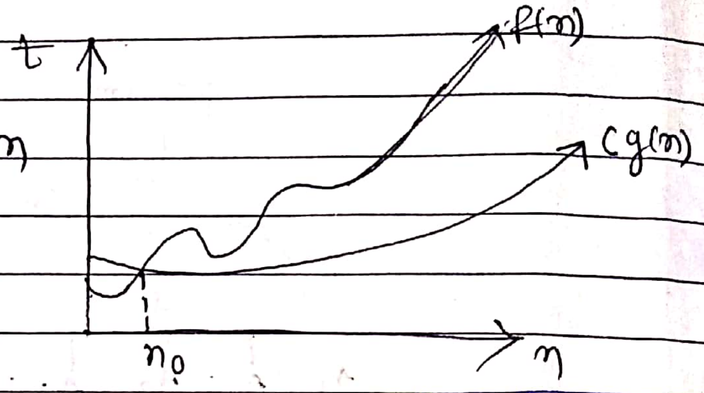
(ii) Big Omega Ω:

ex:

$f(n) = 3n+2$, $g(n) = n$

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

$f(n) \geq c\,g(n)$.



$\boxed{3n+2 \geq cn}$ $c=1$

$n_0 \geq 1$

$\boxed{f(n) \geq c\,g(n),\ n \geq n_0 \quad c>0,\ n_0 \geq 1.}$

$\Rightarrow 3n+2 = \Omega(n)$

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

$\downarrow$

$\log n$

$\downarrow$

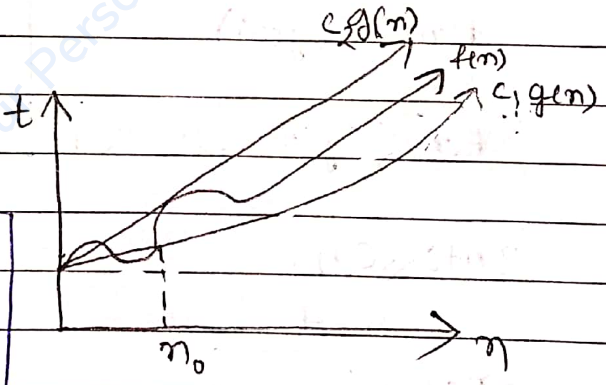
$(\log\log n)$

take always closest lowest bound

here closest bound is 'n'.

(iii) Big theta Θ:

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



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

$c_1, c_2 > 0$

$n \geq n_0$

$\boxed{n_0 \geq 1}$

$3n^2 + n + 1 = \Theta(n^2)$ (Always take leading term)

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

Interested.

O	Ω	θ
Worst case time or the Upper bound.	Best case	Average case

→ Average (θ) case is used when worst case and best case ~~is same.~~ both are same.

ex: array

5	7	2	3	6	9	20	11

then find x in this array by linear.

→ In Best case time $= \Omega(1)$ - (found out x in 1st index).
In worst case time $= O(n)$ - (if the size of array 'n') ~~for~~ then found out x in n^{th} position)

In average case time taken $= \theta(n/2) = \theta(n)$.

• Time complexity Analysis of iterative programs =

→ Algorithm are two types =

Algo
- Iterative
- Recursive

Iterative:
```
A()
{
   for i=1 to n
      max(a,b)
}
```

Recursive:
```
A(n)
{ if ( )
  A(n/2)
}
```

→ Any program that can be ~~to~~ written using iteration ~~could~~ Could be written using Recursion.

→ Any program that can be written using recursion could be written using iteration.

~~→ Any program that not contain iteration and Recursion~~

→ If there is no iteration and recursion inside the program you need not worry about ~~the~~ time. for such program time – O(1).

- ~~some~~ Example of Iterative program =

① ~~A()~~

```
A()
{ int i;
   for(i=1 to n)
      printf("ravi");
}
```

→ Time complexity O(n).
(ravi ~~exe~~ printed n times)

②

```
A()
{ int i, j;
   for(i=1 to n)
      for(j=1 to n)
         printf(ravi);
}
```

→ Time complexity O(n^2).

③

```
A()
{ i=1, s=1;
  While(s<=n)
  { i++;
    s=s+i;
    pf("ravi");
  }
}
```

s = 1 3 6 10 15 21 ... n
i = 1 2 3 4 5 6 .... K

sum of natural no, $\frac{K(K+1)}{2} > n$

$\frac{K^2+K}{2} > n$

$K = O(\sqrt{n})$

→ Time complexity = O($\sqrt{n}$).

④

~~A()~~ A()

{ i = 1; $k = \sqrt{n}$

for (i = 1 ; $\boxed{i^2 <= n}$; i++)

pf("ravi");

}

→ Time complexity − $O(\sqrt{n})$.

⑤

A()

{ int i, j, k, n;

for (i = 1 ; i <= n ; i++)

{ for (j = 1 ; j <= i ; j++)

{ for (k = 1 ; k <= 100 ; k++)

{ pf("Ravi");

}

}

}

}

→ i = 1	i = 2	i = 3	i = 4
j = 1 times	j = 2 times	j = 3	j = 4 times
K = 100 times	K = 2 * 100 times	K = 3 * 100	K = 4 * 100

i = n

j = n times

K = n * 100

100 + ~~200~~ 2 * 100 + 3 * 100 + 4 * 100 + 5 * 100 + ... + n * 100

$\Rightarrow 100 (1 + 2 + 3 + 4 + 5 + \cdots + n)$

$\Rightarrow 100 \frac{n(n+1)}{2}$

$\Rightarrow 100 \left(\frac{n^2+n}{2}\right) \Rightarrow$ time complexity = $O(n^2)$.

6

```
A()
{ int i,j,k,n;

  for(i=1; i<=n; i++)
  {
    for(j=1; j<= i²; j++)
    {
      for (k=1; k= n/2; k++)
      {
        pf("Ravi");
      }
}}}
```

→

$i=1$	$i=2$	$i=3$	$i=4$	$i=5$
$j=1$ time	$j=4$	$j=9$	$j=16$	$j=25$ times
$k=n/2*1$	$k=n/2*4$	$k=n/2*9$	$k=n/2*16$	$k=n/2*25$

$i=n$
$j=n^2$ times
$k=n/2*n^2$

$\rightarrow n/2 + n/2*4 + n/2*9 + n/2*16 + n/2*25 + \dots + n/2*n^2$

$\rightarrow n/2 \left(1+2^2+3^2+4^2+\dots+n^2\right)$

$\rightarrow n/2 \left(\frac{n(n+1)(2n+1)}{6}\right)$ ← A.P.

$$\boxed{\begin{aligned} f(n) &= n^k + n^{k-1}, \dots \\ &= O(n^k) \end{aligned}}$$

$\rightarrow \frac{1}{12} n(n^2+n)(2n+1)$

$\rightarrow 1/12\, n(2n^3+n^2+2n^2+n)$

$\rightarrow 1/12 (2n^4+3n^3+n^2) \rightarrow$ Time complexity $- O(n^4)$.

Date
Page

⑦

10 ($\log_{10} n$)
4 ($\log_4 n$)
3 ($\log_3 n$)

```
A()
{ for(i=1; i<n; i=i*2)
    pf(ravi);
}
```

→ $i = 1, 2, 4, 8, \ldots n$
$2^0 \quad 2^1 \quad 2^2 \quad 2^3, \ldots \quad 2^K$

$\frac{n}{2}+1$
$\frac{n+3}{2}$

$2^K = n$
$K = \log_2 n$

Time complexity or time taken to execute $= O(\log_2 n)$

⑧

```
A()
{
   int i,j,k;
   for(i=n/2; i<=n; i++)        — Independent loop — n/2
     for(j=1; j<=n/2; j++)      — n/2
       for(k=1; k<=n; k=k*2)    — log2 n
          pf("ravi");
}
```

→ $n/2 * n/2 * \log_2 n$

→ $\frac{n^2}{4} \log_2 n \rightarrow O(n^2 \log_2 n) \Leftarrow$ Time complexity.

⑨

```
A()
{ int i,j,k;
   for(i=n/2; i<=n; i++)       — n/2
     for(j=1; j<=n; j=2*j)     — log2 n.
       for(k=1; k<=n; k=k*2)   — log2 n
          pf("ravi");
}
```

~~ckkk~~ → $n/2 \ (\log_2 n)^2 \rightarrow O(n(\log_2 n)^2) \leftarrow$ Time complexity
(removed constants)

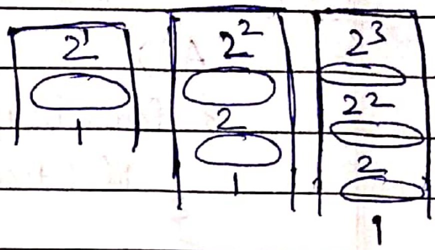
Date
Page

(10)

assume $n \geq 2$

```
A()
{ while(n>1)
   { n = n/2
   }
}
```

3 — $\log_3 n$
5 — $\log_5 n$
⋮



$n = 2^k$

$k = \log_2 n$

Time complexity — $O(\log_2 n)$.

(11)

```
A()
{ for (i=1; i<=n; i++)   n
   { for (j=1; j<=n; j=j+i)
       { pf("ravi"); }
   }
}
```

→ $i=1$	$i=2$	$i=3$	$i=k$
$j=1$ to n ~~times~~	$j=1$ to n	$j=1$ to n ..	$j=1$ to n ...
(ravi printed) — n times	$n/2$ times	$n/3$ times	n/k times

$i=n$
$j=1$ to n
1 times (Ravi printed)

$\rightarrow n + n/2 + n/3 + n/4 + \cdots + n/k + \cdots + n/n$

$\rightarrow n(1 + 1/2 + 1/3 + 1/4 + \cdots + 1/n)$

$\rightarrow n \log n$

→ time complexity — $O(n\log n)$.

12

A()

```
{ int n = 2^(2^K);
  for(i=1; i<=n; i++)      — n
  {
    j=2
    while (j<=n)
    {
      j=j^2;
      pf("Ravi");
    }
  }
}
```

→ $K=1$	$K=2$	$K=3$
$n=4$	$n=2^4$	$n=2^8$
$j=2,4$	$j=2^1,2^2,2^4$	$j=2^1,2^2,2^4,2^8$
$n*$ 2 times	$n*$ 3 times	$n*$ 4 times

→ $\boxed{n*(K+1)}$

$n=2^{2^K}$

$\log_2 n = 2^K$

$K = \log_2 \log_2 n$

→ $n(\log\log n + 1)$

→ Time complexity of this algorithm — $O(n\log\log n)$.

Date
Page

- Time Complexity Analysis of recursive program =

(1)
```
A(n)
{ if(n>1)
    return (A(n-1));
}
```

By using Back-Substitution method ~~you~~ we can find the time complexity of any ~~Algo that~~ recursion program

$\rightarrow$

$$T(n) = 1 + T(n-1) \; ; \; n>1$$
$$= 1 \; ; \; n=1$$

Back Substitution = (method)

$$T(n) = 1 + T(n-1) \quad \text{---} (1)$$
$$T(n-1) = 1 + T(n-2) \quad \text{---} (2)$$
$$T(n-2) = 1 + T(n-3) \quad \text{---} (3)$$

put equ (2) & (3) into equ (1) =

$$T(n) = 1 + 1 + T(n-2)$$
$$= 1 + 1 + 1 + T(n-3)$$
$$= 3 + T(n-3)$$
$$\vdots$$
$$= K + T(n-K)$$

$n-K=1$
$K=n-1$

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

$\boxed{T(n) = O(n)} \rightarrow$ time complexity.

Date
Page

(2)

$$T(n) = n + T(n-1) \quad ; n > 1$$
$$= 1 \quad ; n = 1$$

find Time complexity

By using back Substitution method.

$T(n) = n + T(n-1)$ — (i)

$T(n-1) = (n-1) + T(n-2)$ — (ii)

$T(n-2) = (n-2) + T(n-3)$ — (iii)

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

$$\boxed{\begin{aligned} n-(k+1) &= 1 \\ n-k-1 &= 1 \\ k &= n-2 \end{aligned}}$$

$$= n + (n-1) + (n-2) + \cdots + (n-(n-2)) + T(1)$$
$$= n + (n-1) + (n-2) + \cdots + 2 + 1 \rightarrow A.P$$
$$= \frac{n(n+1)}{2} = \frac{n^2+n}{2}$$ (here most significant term n^2)

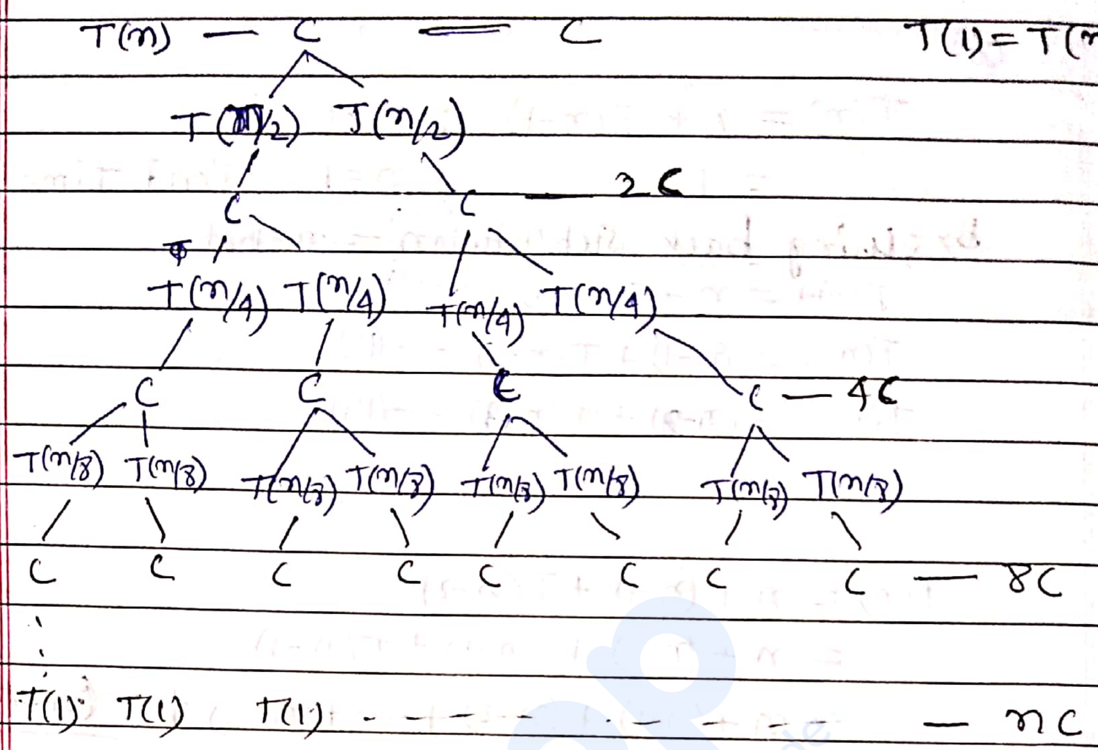
So, time complexity – $O(n^2)$.

(3) Recursion tree method

$$T(n) = 2T(n/2) + C ; n > 1$$
$$= C \quad n = 1$$

find time Complexity using Recursion tree method –

→ $T(n) = 2T(n/2) + C$.



$T(1) = T(n/n)$

$\rightarrow c + 2c + 4c + 8c + \cdots + nc$

$\rightarrow c(1 + 2 + 4 + 8 + \cdots + n)$

$\boxed{\text{let } n = 2^K}$

$\rightarrow c(2^0 + 2^1 + 2^2 + 2^3 + \cdots + 2^K)$ — GP

$\rightarrow c \dfrac{(2^{K+1} - 1)}{(2-1)}$

$\rightarrow c(2^{K+1} - 1)$

$\rightarrow c(2^K \cdot 2 - 1)$

$\rightarrow c(2n - 1)$

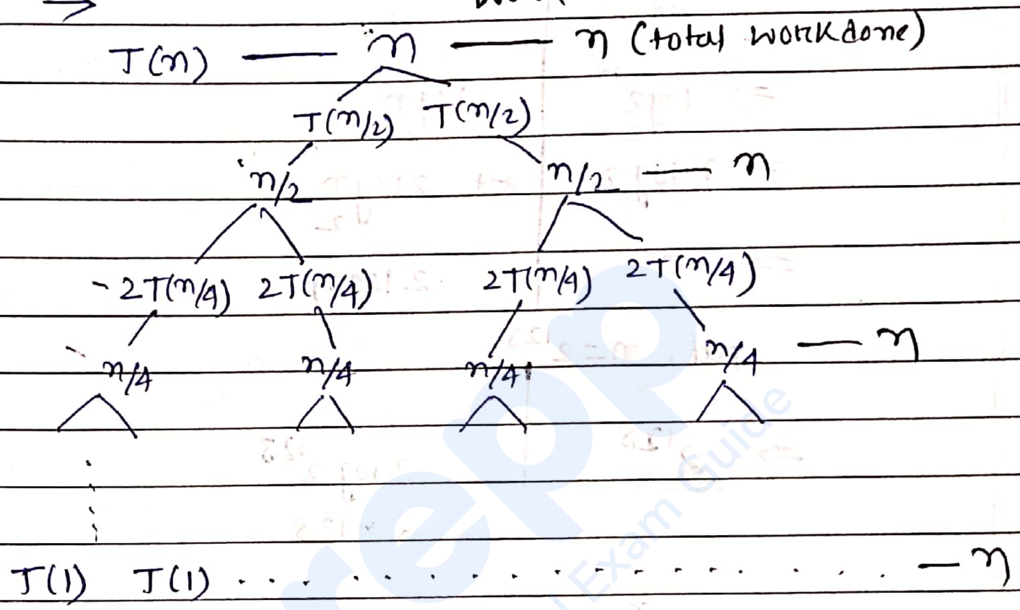
$\Rightarrow O(n) \rightarrow$ Time complexity.

④ $T(n) = 2T(n/2) + n \; ; \; n > 1$

$\quad\quad = 1 \quad\quad\quad ; \; n = 1$

find time complexity using Recurcsion tree method =

→ Work done each level –



→ $n/2^0 \to n/2^1 \to n/2^2 \to n/2^3 \to \cdots \to n/2^k$ $\quad\quad \boxed{n = 2^k}$

$k = \log_2 n$

→ $n/2^0 + n/2^1 + n/2^2 + n/2^3 + \cdots + n/2^k$

→ $n\left(1/2^0 + 1/2^1 + 1/2^2 + 1/2^3 + \cdots + 1/2^k\right)$

→ $n\,(k+1)$

→ $n\,(\log_2 n + 1)$

So, here, Time complexity is – $O(n \log_2 n)$.

(5) ~~Easiest way to solve recursion~~ — $\boxed{\text{~~Master's Theorem~~}}$ (~~time complexity~~)

• Comparing various functions to analyse time complexity —

(1) ex-1

2^n	n^2
$\Rightarrow \log_2 2^n$	$\log_2 n^2$
$\Rightarrow n\log_2 2$	$2\log_2 n$
$\Rightarrow n$	$2\log n$

put, $n = 2^{128}$

$\Rightarrow 2^{128}$	$2\log 2^{128}$ $2 * 128$

So, 2^n is larger than n^2.

ex-2

n^2	$n\log n$
$\rightarrow \cancel{n} \times n$	$\cancel{n}\log n$
$\rightarrow n$	$\log n$

function n^2 is greater than $\log n$.

ex-3

n	$(\log n)^{100}$
$\rightarrow \log n$	$\rightarrow 100\ \log\log n$
$\boxed{n = 2^{2^{10}}}$	
$\rightarrow 2^{10}$	$\rightarrow 100\ \log\log 2^{2^{10}}$
$\rightarrow 1024$	$\rightarrow 1000$

n is larger than $\log n$.

ex-4

$n^{\log n}$ > $n \log n$

$\rightarrow \log(n^{\log n})$ | $\rightarrow \log(n \cdot \log n)$

$\rightarrow \log n \log n$ | $\rightarrow \log n + \log \log n$.

$\boxed{n = 2^{128}}$

$\rightarrow \underline{128 * 128}$ | $\rightarrow 128 + 7$

$\boxed{n^{\log n} > n \log n}$

ex-5

$\sqrt{\log n}$ > $\log \log n$

$\frac{1}{2} \log \log n$ | $\log \log \log n$

$\boxed{n = 2^{2^{10}}}$

$\frac{1}{2} * 10$ | $\log 10$

$= \underline{\underline{5}}$ | $= 3.5$

$\sqrt{\log n} > \log \log n$

ex-6

$n^{\sqrt{n}}$ > $n^{\log n}$

$\rightarrow \log n^{\sqrt{n}}$ | $\rightarrow \log n^{\log n}$

$\rightarrow \sqrt{n} \log n$ | $\rightarrow \log n \log n$.

$\rightarrow \sqrt{n}$ | $\rightarrow \log n$

$\rightarrow \frac{1}{2} \log n$ | $\rightarrow \log \log n$.

$\boxed{n = 2^{128}}$

$\rightarrow \frac{1}{2} * 128$ | $\rightarrow$ 7

Date
Page

ex-7

$$f(n) = \begin{cases} n^3 & 0 < n < 10000 \\ n^2 & n \geq 10000 \end{cases}$$

$$g(n) = \begin{cases} n & 0 < n < 100 \\ n^3 & n > 100 \end{cases}$$

→

	0 – 99	100 – 9999	1000 - - - ∞
f(n)	n^3	n^3	n^2
g(n)	n	n^3	n^3

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

$g(n) > f(n)$

$\boxed{\text{ex-8}}$ $f_1 = 2^n$, $f_2 = n^{3/2}$, $f_3 = n\log n$, $f_4 = n^{\log n}$

2^n	$n^{3/2}$	$n\log n$	$n^{\log n}$
$n\log_2 2$	$3/2 \log n$	$\log n + \log\log n$	$\log n \log n$.
	$\boxed{n = 2^{128}}$		
→ 2^{128}	→ $3/2 * 128$	→ $128 + 7$	→ $128 * 128$

$$\boxed{f_1 > f_4 > f_2 > f_3}$$

→ This is how functions are compared.

$(\log n)^2 \neq \log^2 n$

$\Rightarrow \log n * \log n \neq \log \log n$

- $\boxed{\text{Masters theorem}}$ = (to find time complexity of recursion program easily)

$$\boxed{T(n) = aT(n/b) + \Theta(n^K \log^p n)}$$

$a \geq 1$, $b > 1$, $K \geq 0$ and p is real Number.

i) if $a > b^K$, then $T(n) = \Theta(n^{\log_b^a})$

ii) if $a = b^K$

a) if $p > -1$, then $T(n) = \Theta(n^{\log_b^a} \log^{p+1} n)$

b) if $p = -1$, then $T(n) = \Theta(n^{\log_b^a} \log \log n)$

c) if $p < -1$, then $T(n) = \Theta(n^{\log_b^a})$

iii) if $a < b^K$.

a) if $p \geq 0$; then $T(n) = \Theta(n^K \log^p n)$

b) if $p < 0$; then $T(n) = \cancel{\Theta}\ O(n^K)$.

$\boxed{\text{ex-1}}$

$T(n) = 3T(n/2) + n^2$;

→ after compare with Masters equation —

here $a = 3$, $b = 2$, $K = 2$, $p = 0$

$$\begin{array}{ccc} a & & b^K \\ \parallel & & \parallel \\ 3 & < & 2^2 \end{array}$$

iii) a)

$T(n) = \Theta(n^2 \log^0 n)$

$$\boxed{T(n) = \Theta(n^2)}.$$

$\boxed{Ex-2}$

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

→ After compare with Masters equation,
$a=4$, $b=2$, $K=2$, $p=0$

$a = 4$ | $b^K = 2^2$

$$\boxed{a = b^K}$$

ii) → a)

$$T(n) = \theta\left(n^{\log_2^4} \log^{0+1} n\right)$$
$$= \theta\left(n^2 \log n\right)$$

$\boxed{Ex-3}$

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

→ After compare with masters equation -

$a=1$, $b=2$, $K=2$, $p=0$

$a=1$ $b^K=4$

$$\boxed{a < b^K}$$

iii) a) $T(n) = \theta\left(n^K \log^p n\right)$
$$= \theta(n^2)$$

$\boxed{Ex-4}$

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

↳ In this case master theorem can't be apply.

$\boxed{\text{Ex-5}}$

$T(n) = 16T(n/4) + n$

$\rightarrow$ $a=16$, $b=4$, $K=1$, $P=0$

$a=16 > b^K=4$

(i) $T(n) = \theta(n^{\log_b^a})$
$= \theta(n^2)$

$\boxed{\text{Ex-6}}$ $T(n) = 2T(n/2) + n\log n$.

$\rightarrow$ $a=2$, $b=2$, $K=1$, $P=1$

$a=2 = b^K=2$

$\boxed{a=b^K}$

ii) a)

$T(n) = \theta(n^{\log_b^a} \log^{P+1} n)$

$= \theta(n\log^2 n)$

$\boxed{\text{Ex-7}}$

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

$= 2T(n/2) + n\log^{-1} n$.

$\rightarrow$ $a=2$, $b=2$, $K=1$, $P=-1$

$\boxed{a=b^K}$

ii) b) $T(n) = \theta(n^{\log_b^a} \log\log n)$

$= \theta(n\log\log n)$

$\boxed{\text{Ex-8}}$

$T(n) = 2T(n/4) + n^{0.51}$.

$\rightarrow$ $a=2,\ b=4,\ k=0.51,\ p=0$

$a=2 < b^k = 4^{0.51}$

iii) a) $T(n) = \theta(n^k \log^p n)$
$T(n) = \theta(n^{0.51})$

$\boxed{\text{Ex-9}}$

$T(n) = 0.5T(n/2) + 1/n$ ✗

$\rightarrow$ $a = 0.5,\ b=2,\ k=-1,\ p=0$ (not possible using masters theorem).

$a = 0.5 \quad b^k = 2^{-1} = 1/2 = 0.5$

$\boxed{\cancel{a=b^k}}$ a should be $\boxed{a \geq 1}$

$\cancel{\text{iii) a) } T(n) \neq \theta(n^{\log_b a}/\log^{p+1} n)}$
$\cancel{= \theta(n^{\log_2 0.5}/\log n)}$

$\boxed{\text{Ex-10}}$ $T(n) = 6T(n/3) + n^2 \log n$.

$\rightarrow$ here, $a=6,\ b=3,\ k=2,\ p=1$

$\boxed{a < b^k}$

iii) a)
$T(n) = \theta(n^k \log^p n)$
$= \theta(n^2 \log n)$

Ex-11

$T(n) = 64\,T(n/8) - n^2 \log n$. ✗

→ here we can't apply masters theorem.
for "–" sign.

Ex-12

$T(n) = 7T(n/3) + n^2$

→ $a=7,\ b=3,\ k=2,\ p=0$

$\boxed{a < b^k}$

iii) a) $T(n) = \theta(n^2)$

Ex-13 $T(n) = 4T(n/2) + \log n$.

→ $a=4,\ b=2,\ k=0,\ p=1$

here, $\boxed{a > b^k}$

i) $T(n) = \theta(n^{\log_b a})$
$= \theta(n^2)$

Ex-14

$T(n) = \sqrt{2}\,T(n/2) + \log n$

$a=\sqrt{2},\ b=2,\ k=0,\ p=1$

here,

$\boxed{a > b^k}$

i) $T(n) = \theta(n^{\log_b a})$

$= \theta(n^{\log_2 \sqrt{2}})$

$= \theta(\sqrt{n})$;

$\boxed{Ex-15}$

$$T(n) = 2T(n/2) + \sqrt{n}$$

$\rightarrow$ $a=2,\ b=2,\ k=1/2,\ p=0$

here,

$$\boxed{a > b^k}$$

i) $T(n) = \theta(n^{\log_b a})$

$= \theta(n^{\log_2 2})$

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

$\boxed{Ex-16}$

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

$\rightarrow$ $a=3,\ b=2,\ k=1,\ p=0$

here,

$$\boxed{a > b^k}$$

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

$\boxed{Ex-17}$

$$T(n) = 3T(n/3) + \sqrt{n}.$$

$\rightarrow$ $a=3,\ b=3,\ k=1/2,\ p=0$

here,

$$\boxed{a > b^k}$$

i) $T(n) = \theta(n^{\log_3 3})$

$= \theta(n).$

Ex-18

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

$\rightarrow$

$a=4,\ b=2,\ k=1,\ p=0$

here, $\boxed{a > b^k}$

i) $T(n) = \theta(n^{\log_2 4})$

$= \theta(n^2).$

Ex-19

$T(n) = 3T(n/4) + (n\log n).$

$\rightarrow a=3,\ b=4,\ k=1,\ p=1$

here, $\boxed{a < b^k}$

iii) a)

$T(n) = \theta(n^k \log^p n)$

$= \theta(n^1 \log^1 n)$

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

Analysis space complexity of iterative and recursive Algorithm =

Algo

iterative   recursive

- **Space Complexity for Iterative program =**

① Algo (A, 1, n)
{
int i; —— (1)
for (i=1 to n)
A[i] = 0;
}

→ space complexity = O(1)

Algo (A, 1, n)
{
int i, j = 10; —— (2)
for (i=1 to j)
A[i] = 0;
}

→ space complexity = O(1)

② Algo (A, 1, n)
{
int i; → (n+1)
creat B[n];
for (i=1 to n)
B[i] = A[i];
}

→ Space complexity – O(n)

③ Algo (A, 1, n)
{ creat B[n, n] —— n^2
int i, j; —— 2
for (i=1 to n)
for (j=1 to n)
B[i, j] = A[i]
}

(n^2+2)

→ here space complexity – $O(n^2)$.

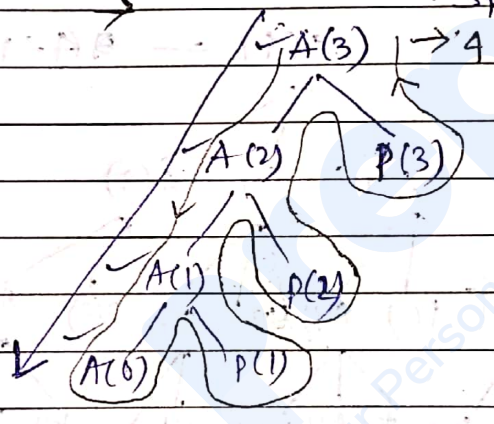
- Space complexity of recursive (Algorithm) program –

→ When ~~the executed program~~ no. of statement less inside the program then use tree method.

ex- 1

```
A(n)            — (n+1)
{ if(n>=1)
   {
     A(n-1);
     pf(n);
   }
}
```

Space complexity = O(n)

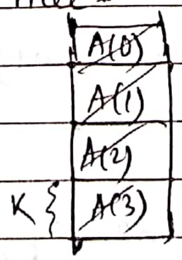


Output = 1 2 3

for A(3) function called – 4 times.

for A(n) function called – n times.

→ Space complexity is depth of the tree.



$n-(n+1)K$
$- O(kn)$
$- O(n)$

Time complexity = recursive equation –

$$T(n) = \begin{cases} 1 & n=0 \\ T(n-1)+1 & ; n \geq 1 \end{cases}$$

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

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

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

$\vdots$

$T(n) = T(n-K) + K$

$= T(n-n) + n$

$= T(0) + n$

$= 1+n$

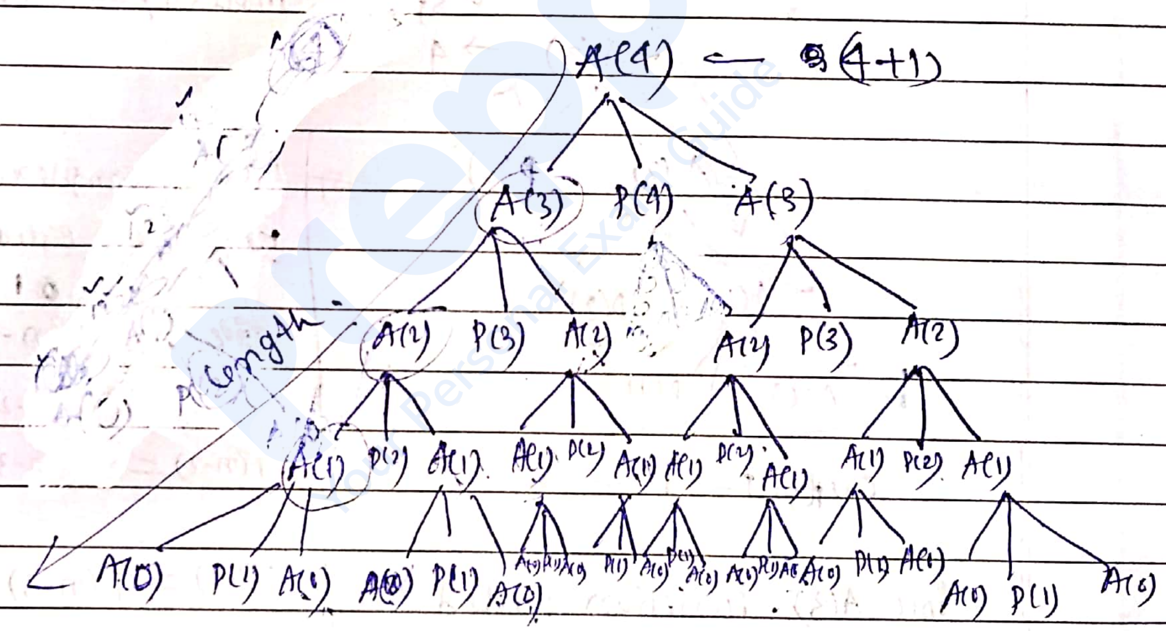
$T(n) = 1+n$

$= O(n)$.

ex-② (find Time & space Complexity)

```
A(n)  ← (n+1)
{
  if (n≥1)
  {
    1. A(n-1);
    2. Pf(n);
    3. A(n-1);
  }
}
```

→ $n = 4$ A(4)



here Space Complexity $= O(n+1)$
$= \boxed{O(n)}$.

output = 1 2 1 3 1 2 1 4 1 2 1 3 1 2 1

$A(4) = 31 = 2^{4+1} - 1$
$A(3) = 15$ (times function call) $= 2^{3+1} - 1$
$A(2) = 7 = 2^{2+1} - 1$
$A(1) = 3 = 2^{1+1} - 1$
:
$A(n) = 2^{n+1} - 1$ (for n value function called $n+1$ times)

Time complexity –

Recursive equation =

$$T(n) = \cancel{2} T(n-1) + 1 + T(n-1)$$
$$= 2T(n-1) + 1 \quad ; \quad n \geq 1$$
$$= 1 \quad ; \quad n = 0$$

$$T(n) = 2T(n-1) + 1 \quad \text{—(i)}$$
$$T(n-1) = 2T(n-2) + 1 \quad \text{—(ii)}$$
$$T(n-2) = 2T(n-3) + 1 \quad \text{—(iii)}$$

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

$$= 2^n T(0) + 2^{n-1} + 2^{n-2} + \cdots + 1 \qquad \boxed{\begin{matrix} n-K=0 \\ K=n \end{matrix}}$$
$$= 2^n \cdot 1 + 2^{n-1} + 2^{n-2} + \cdots + 1$$
$$= 2^n + 2^{n-1} + 2^{n-2} + \cdots + 2^0 \quad \text{— GP}$$

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

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

Sorting Techniques

www.gatenotes.in

• Insertion sort algorithm and analysis :-

```
Insertion-sort (A)
{
  for(j=2 to A.length)
  {
     Key = A[j];    // insert A[j] into stored sequence
                       A[1....j-1]
     i = j-1
     while(i>0 and A[i] > Key)
     {  A[i+1] = A[i].
        i = i-1  }
     } A[i+1] = Key  }
  }
```

Working Procedure :-

→ Array

1	2	3	4	5
2	9	6	5	7
i	j			

Key = 9

(1)

1	2	3	4	5
2	9	6	5	7
	i	j		

Key = 6

(2)

1	2	3	4	5
2	6	9	5	7
i		j		

Key = 6

(3)

1	2	3	4	5
2	6	9	5	7
		i	j	

Key = 5

(4)

1	2	3	4	5
2	5	6	9	7
			i	j

Key = 7

(5)

1	2	3	4	5
2	5	6	7	9

array (sorted)

- Time Complexity in worst case = $O(n^2)$

5	4	3	2	1
1	2	3	4	5

→ When reversed array

Index		Comparisons		movement		
2	–	1	+	1	= 2	= 2(1)
3	–	2	+	2	= 4	= 2(2)
4	–	3	+	3	= 6	= 2(3)
5	–	4	+	4	= 8	= 2(4)
⋮						
n	–	$(n-1)$	+	$(n-1)$	$= 2(n-1)$	

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

$$= 2\left(1+2+3+4+\cdots+(n-1)\right) \quad (A.P) \qquad \boxed{\frac{n(n+1)}{2}}$$

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

$$= n^2 - n$$

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

- Time Complexity in Best case = ~~$O(n)$~~ $\Omega(n)$.

1	2	3	4
1	2	3	4

→ when array ~~at~~ already sorted.

Index		Comparisions		movement	
2	–	1	+	0	= 1
3	–	1	+	0	= 1
4	–	1	+	0	= 1
⋮					⋮
n	–	1	+	0	= 1

$$T(n) = 1+1+1+1+\cdots+1 = \Omega(n-1) = \Omega(n).$$

- Space Complexity = O(1)

Key, i, j

(need only 3 variable)

→ When we need constant space to sort any given list, such algo also call Inplace Algo.
So, Insertion sort also called Inplace Algo.

When used	Comparisons	movement	
Binary Search	$O(\log n)$	n	$= O(n^2)$ (T.C)
double linked list	$O(n)$	$O(1)$	$= O(n)$ (T.C)

- Merge Sort algorithm and analysis =

MERGE (A, P, q, r) [Merge procedure]
{
$n_1 = q - p + 1;$
$n_2 = r - q;$
let L [1 ... n+1] and R [1 to n_2+1] be new arrays
for (i = 1 to n_1)
L[i] = A[P+i-1]

for (j = 1 to n_2)
R[j] = A[q+j];

L[n_1+1] = ∞;
R[n_2+1] = ∞;
i = 1, j = 1;

for (k = p to r)
    if (L[i] ≤ R[j])
        A[k] = L[i]
        i = i+1;
    else
        A[k] = R[j]
        j = j+1;
}

ex:

	p 1	2	3	q 4	q+1 5	6	7	r 8
→ A	1	5	7	8	2	4	6	9

L	1	5	7	8	∞
	i				

R	2	4	6	9	∞
	j				

— $O(n)$

	1	2	3	4	5	6	7	8
A	1	2	4	5	6	7	8	9

— $O(n+m) - O(n)$

Sorted using (merge sort)

Total time taken by merge sort = $O(n)$.
Space complexity = $O(n)$.

⇒ ~~Merge procigun is also called~~
~~Merge procigin is also called out of place proci~~

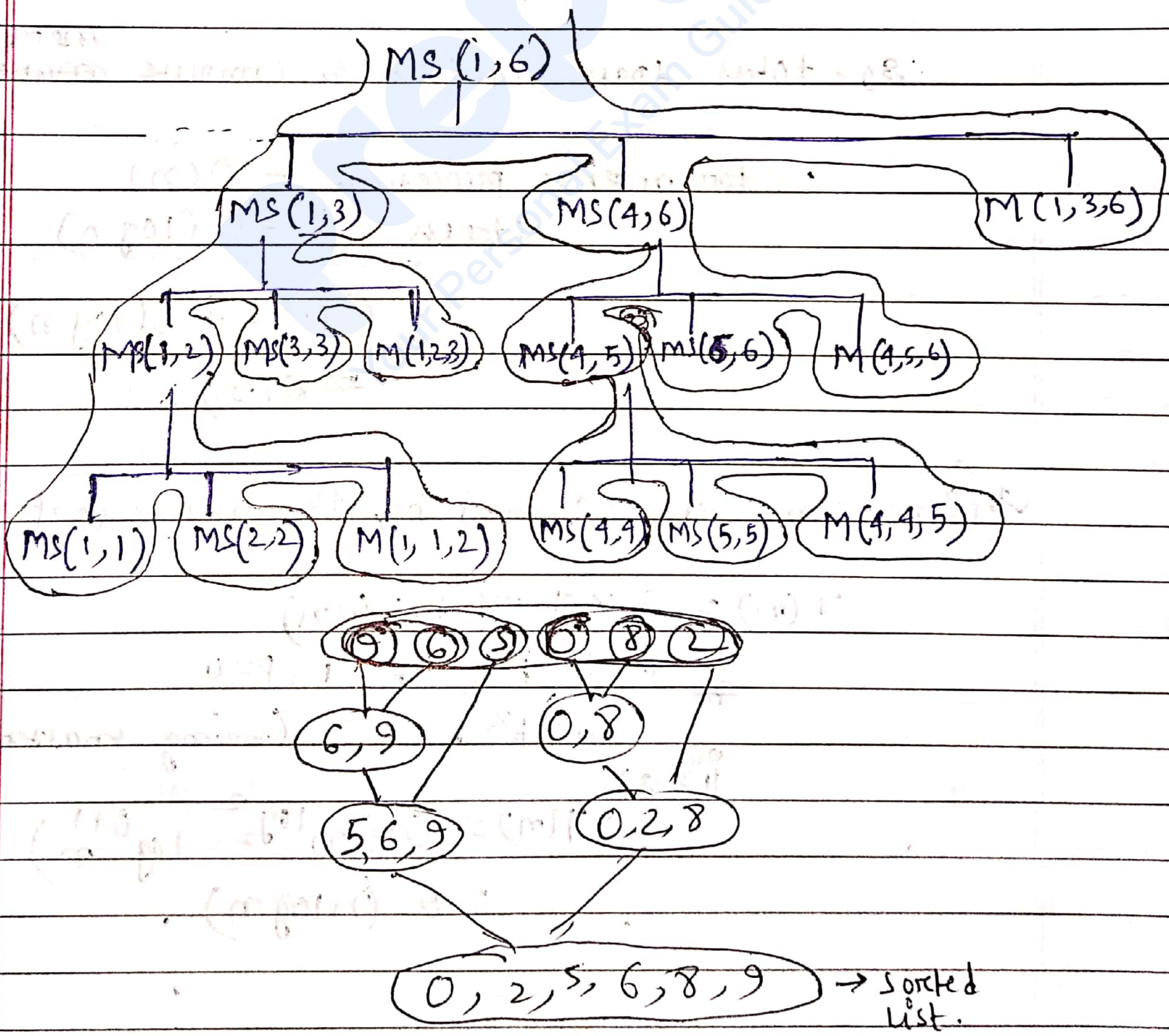
→ Merge procedure is also called out of place procedure.

[Merge-sort]

```
merge-Sort(A,p,r)  — T(n)
{
    if p<r
    q = ⌊(p+r)/2⌋
    merge-sort(A,p,q)      — T(n/2)
    merge-sort(A,q+1,r)    — T(n/2)
    merge(A,p,q,r)         — O(n)
}
```

Ex:

	1	2	3	4	5	6
A	9	6	5	0	8	2
	p					r



Space Complexity required by the merge sort – $O(n)$

4)	~~4~~	~~8 8 11 11 13~~
3)	~~3~~	~~7 8 10 14 15~~
2)	~~2~~	~~8 16~~
1)	~~1~~	

6 — 4 level

$n - (\lceil \log n \rceil + 1)$ lines.

size of stack $= (\lceil \log n \rceil + 1) K$

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

So, total space required to complete ~~marge~~ merge sort:

for merge procedure — $O(n)$
stack — $O(\log n)$

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

Time complexity required by the merge sort — $O(n \log n)$

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

$a=2,\ b=2,\ K=1,\ P=0$

$\boxed{a = b^K}$ (using masters theorem)

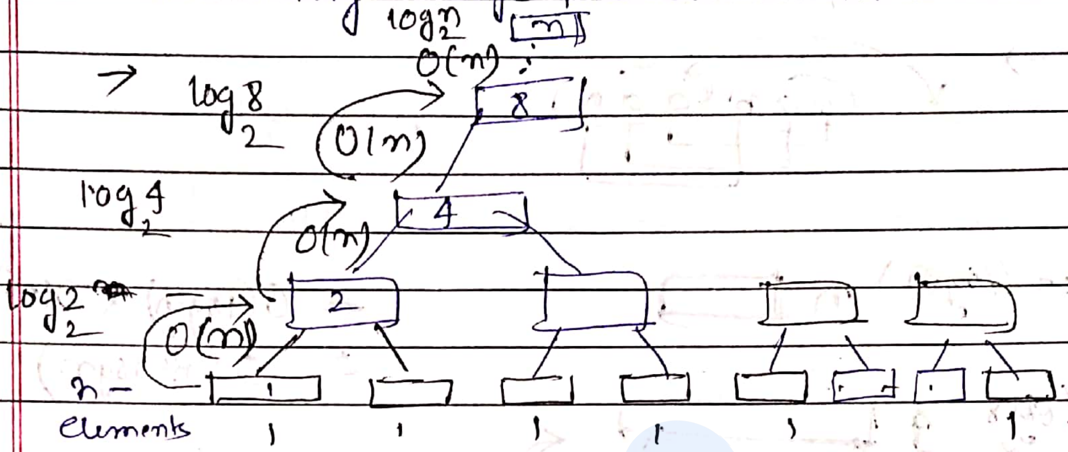
ii) a)

$T(n) = \theta\left(n^{\log_2 2} \log^{0+1} n\right)$
$= \theta(n \log n)$.

Date
Page

Q-1 (2-way mergeing)

Given 'n' elements, merge them into one sorted list using merge procedure. What is time complexity -

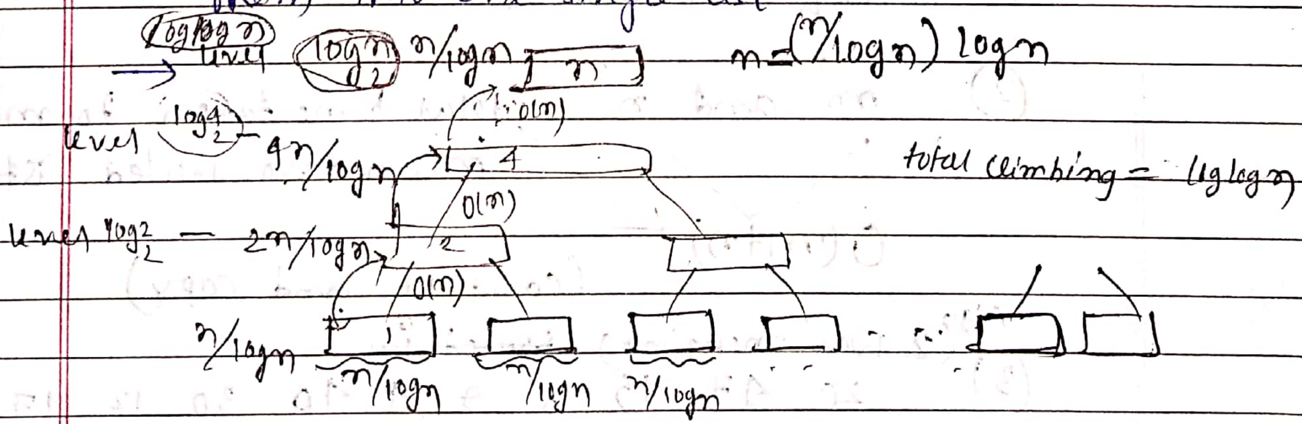


Total time $= O(n) * O(\log n)$
$= \boxed{O(n\log n)}$

$O(n) \to$ work done each level.
no. of levels $\to O(\log n)$

Q-2

Given "log n" sorted lists of size "n/log n", what is the total time required to merge them into one single list -



~~total element~~

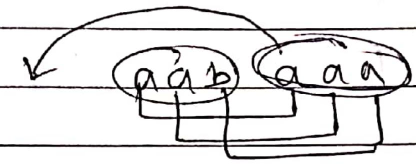
$\log n * n/\log n = n$ element each level.

~~time comp~~ Time complexity $= O(n\log\log n)$.

Q-3

'n' strings each of length "n" are given then what is the time taken to sort them

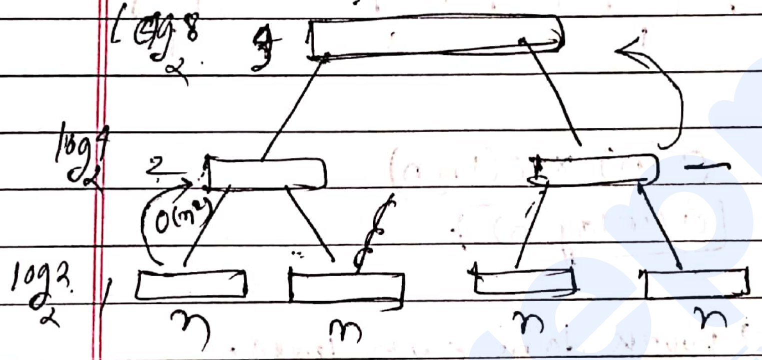
→



uses $(\log n)$ n ▭

$= O(\log n) \times O(n^2)$

$= O(n^2 \log n)$



— $O(n) \times O(n) + O(n^2)$

$= O(n^2)$

$= O(n^2)$ element

Total time Complexity $= O(n^2 \log n)$.

Q-4

① Merge sort uses Divide and Conquer.

② 'm' and 'n' total time taken to merge m, n two sorted list —

$O(m+n)$

(compare and copy)

Algo (2 way merging) ~~Sorted in~~

③ 20 47 15 8 9 4 40 30 12 17

What will be the order of elements after 2nd pass —

→ (20 47 15 8 9 4 40 30 12 17)

(20 47) (8 15) (4 9) (30 40) (12 17)

(8, 15, 20, 47) (4, 9, 30, 40) (12, 17)

↳ After 2^{nd} pass this is the order we get.

• Quick Sort Algorithm =

→ For smaller ~~no. of~~ input, Quick sort run faster compare to merge sort.

→ Quick Sort and merge sort ~~Both~~ follow divide and conquer method.

• Partition Algorithm =

Time taken by the partioning algorithm = $O(n)$.

ex: i j

9	6	5	0	8	2	4	7
6	5	0	9	8	2	4	7
6	5	0	2	8	9	4	7
6	5	0	2	4	9	8	7
6	5	0	2	4	7	8	9

< 7 <

Quick Sort

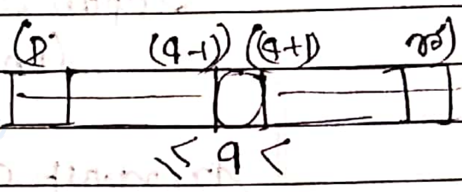
ex: A (p ↓ at 1, r ↓ at 12)

1	2	3	4	5	6	7	8	9	10	11	12
13	19	9	5	12	8	7	4	21	2	6	11

```
PARTIONTION (A, P, r)
{
   x = A[r]
   i = P-1;
   for (j=P to r-1)
   { i=i+1;
     exchange A[i] with A[j]   } ⟹
```

```
if (A[j] ≤ x)
    {
      i = i+1;
      exchange A[i] with A[j]
    }
}
exchang A[i+1] with A[r]
return i+1;
}
```

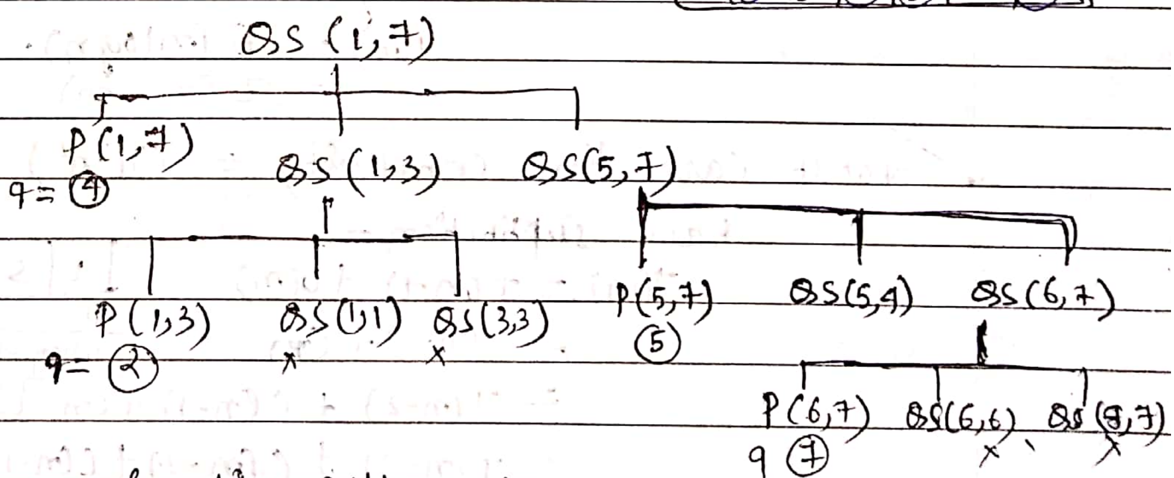
```
QUICKSORT (A, p, r)   — T(n)
{
  if ( p<r)
  {
     q = PARTITION (A, p, r);   — O(n)
     QUICKSORT (A, p, q-1);     — T(n/2)
     QUICKSORT (A, p, q+1);     — T(n/2)
  }
}
```



ex:

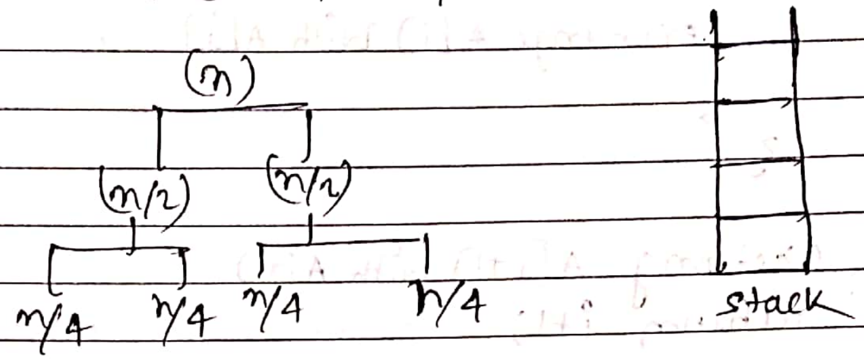
	1	2	3	4	5	6	7
A	5	7	6	1	3	2	4

1	2	3	4	5	6	7



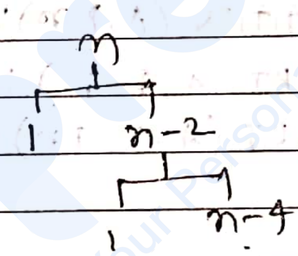
total function call = 13

→ no. of levels in the tree equals to the no. of stack entries required.



in best case
✓ space complexity = $O(\log n)$
(in case if it is balanced)

✓ in worst case space complexity = $O(n)$
(unbalanced)



✓ In best case time complexity = $\Theta(n \log n)$

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

using masters theorem

$T(n) = \theta(n \log n)$.
$= \Omega(n \log n)$

✓ worst case time complexity ≤ $O(n^2)$

back substitution –

$T(n) = T(n-1) + O(n)$
$= T(n-1) + Cn$
$= T(n-2) + C(n-1) + Cn$
$= T(n-3) + C(n-2) + C(n-1) + Cn$.
⋮

1	5	6	7	10

p r

when array in ascending order $T(n) = O(n^2)$

Date
Page

~~else~~

$= c + c2 + c3 + \cdots + cn = c(1+2+\cdots n)$

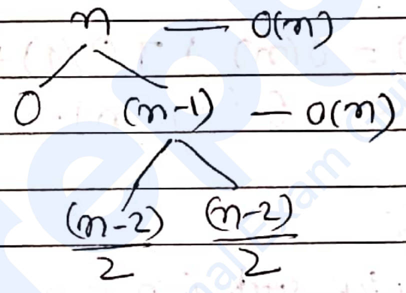
$\cancel{= O(n^2)}$ $\qquad = \frac{n(n+1)}{2}$

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

→ Even input in assending order or desending order ~~to~~, time complexity is $= O(n^2)$.
and also i/p all are same, then time compexity $= O(n^2)$

~~ex:~~

→ best and worst combination –



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

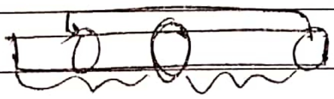
$\leq \cancel{2}\,O(n) + T(n/2)$.

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

$\boxed{\text{Question}}$ – 1

The median of 'n' elements can be found in O(n) time. which one of the following is correct about complexity of quick sort, in which median is selected as pivot ?

→



to found median $= O(n)$
replace $= O(1)$
partition $= O(n)$

$T(n) = O(n) + O(1) + O(n) + 2T(n/2)$ $\qquad - n/2 + n/2$

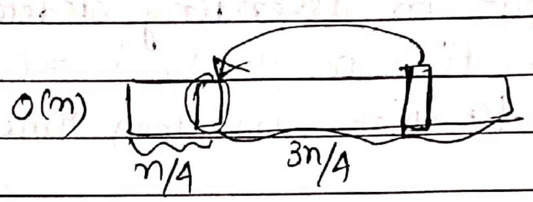
~~using~~ $= O(n) + 2T(n/2)$

using master theorem, $T(n) = \theta(n \log n)$

Question - 2

In quick sort, for sorting 'n' elements, the $(n/4)^{th}$ smallest element is selected as pivot using $O(n)$ time algorithm. What is the worst space complexity of quick sort.

→



$$T(n) = O(n) + O(1) + \underbrace{O(n)}_{\text{partition}} + T(n/4) + T(3n/4).$$

$$T(n) = O(n) + T(n/4) + T(3n/4)$$
$$= \theta(n \log n).$$

1 : 3
1 : 9
1 : 99
1 : 999

$\boxed{\theta(n\log n)}$

Question - 3

Using Quick sort on an algorithm, given i/p

1, 2, 3, . . . n — Time taken T_1

n, n-1, n-2, . . . 1 — " " T_2

What is the relationship between T_1 & T_2.

→ either elements are ascending order or descending order or all equal then all this case time taken $O(n^2)$

$(T_1 = T_2)$.

Date
Page

$\boxed{\text{Question}}$ - 4

parition algo which take $O(n)$ time, we are spliting the problem into two part $\frac{1}{5}n$, $\frac{4}{5}n$, then what is time complexity.

→ $T(n) = O(n) + T(n/5) + T(4n/5)$

$T(n) \leq O(n) + T(4n/5)$

• Introduction to - HEAPS :

	insert	search	Find min	Delete min
unsorted array	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Sorted array	$O(n)$	$O(\log n)$	$O(1)$	$O(n)$
unsorted linked list	$O(1)$	$O(n)$	$O(n)$	$O(n)$
min heap	$O(\log n)$		$O(1)$	$O(\log n)$

→ heap is ~~an~~ datastructure which used optimise time complexity of some of the operation

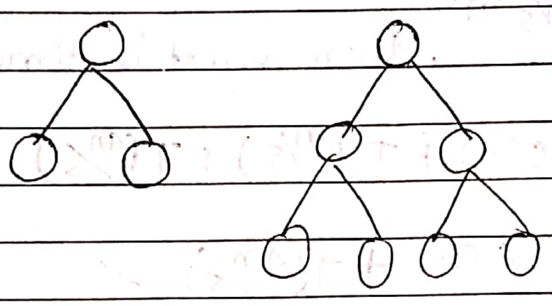
→ heap

min heap   max heap.

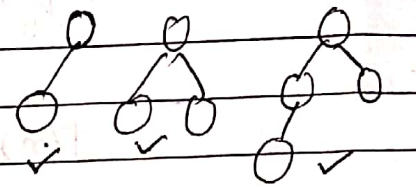
→ using heap we can implement heap sort algorithm.

→ Heap could implement as a binary tree, or 3-ary ~~three tree~~ ; ∴ n-ary tree

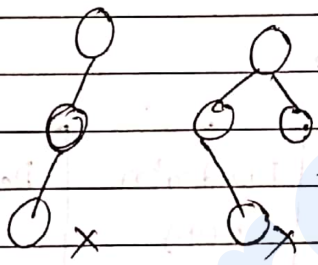
→ Every heap is an almost complete binary tree.



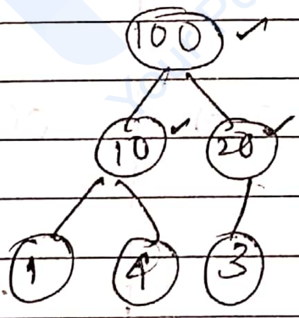
Complete B.T.



almost complete B.T.

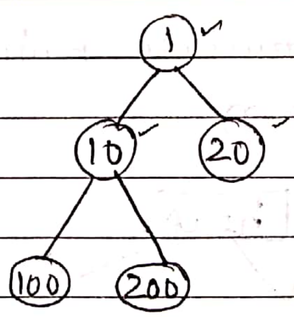


Max-heap :



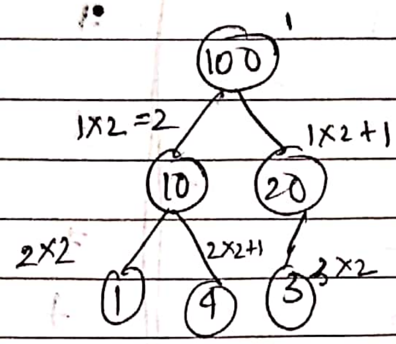
→ all the elements in the root should be grater than leaf.

Min-heap :



→ minimum element will present ~~o~~ in the root.

<u>max heap</u> = Store complete binary tree in a array =



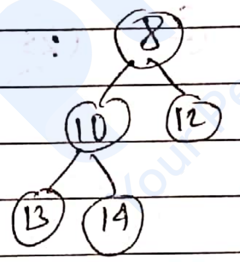
array

1	2	3	4	5	6
100	10	20	1	4	3

left child(i) = $2 \times i$
Right " (i) = $2 \times i + 1$.
parent(i) = $\lfloor i/2 \rfloor$

→ if the array in assending order —, then it is already min heap.

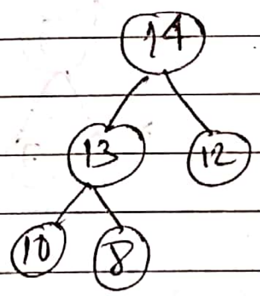
8	10	12	13	14



(root element always less than it child)

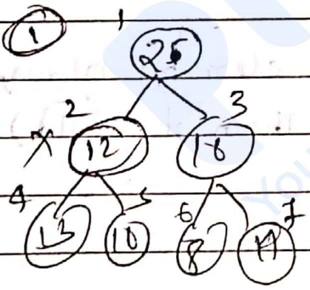
→ if the array in decending order — then it is already max heap.

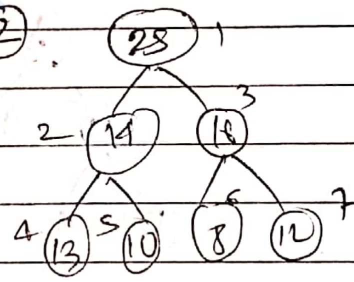
14	13	12	10	8

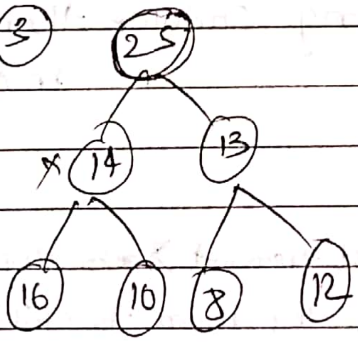


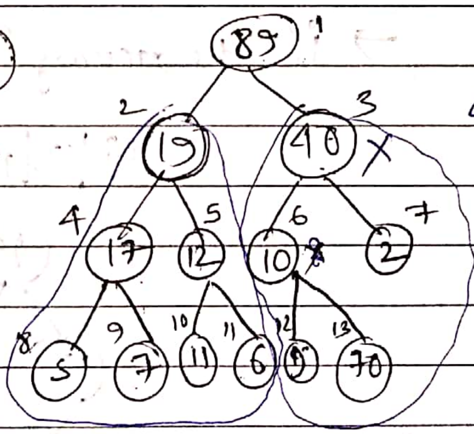
(root element ~~[illegible]~~ always greater than its child)

	Array	Array length	Array heap size
①	25, 12, 16, 13, 10, 8, 14	7	1
②	25, 14, 16, 13, 10, 8, 12	7	7 (Max heap)
③	25, 14, 13, 16, 10, 8, 12	7	1
④	25, 14, 12, 13, 10, 8, 16	7	2
⑤	14, 13, 12, 10, 8	5	5 (max h)
⑥	14, 12, 13, 8, 10	5	5 (max h)
⑦	14, 13, 8, 12, 10	5	5 (max h)
⑧	14, 13, 12, 8, 10	5	5 (max heap)
⑨	89, 19, 40, 17, 12, 10, 2, 5, 7, 11, 6, 9, 70	13	2

①


②


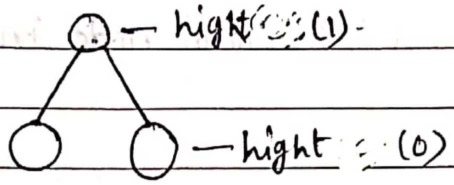
③


⑨


40<70
not follow
Max heap
property

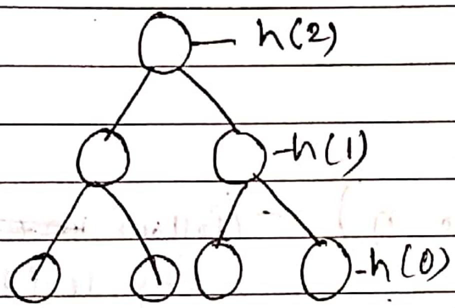
Date
Page

Some properties of complete binary tree =

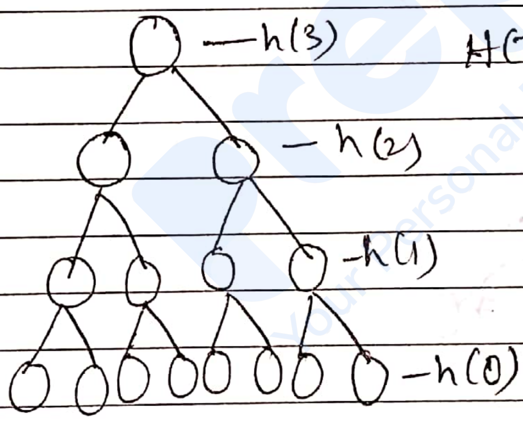


hight of a tree = hight of root.

$H(t) = 1$



$H(T) = 2$



$H(T) = 3$

Hight	1	2	3	4	....	h
max node	3	7	15	31	....	$(2^{h+1}-1)$

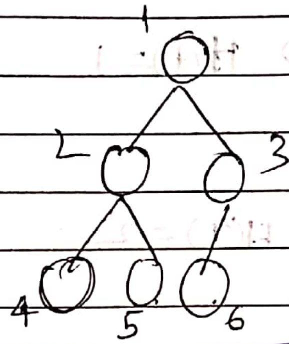
- max no.of node in complete binary tree $= (2^{h+1}-1)$

($h \rightarrow$ hight)

- 'n' nodes inside a complete or almost complete binary tree, ~~what is~~ then hight of tree $= \lfloor \log n \rfloor$

→ hight of any binary heap is = $\lfloor \log n \rfloor$

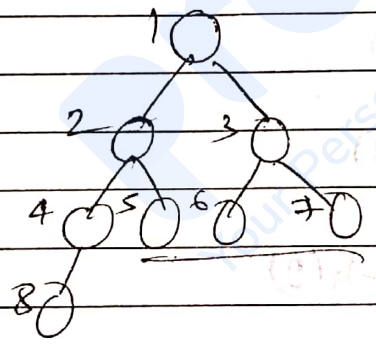
($n \rightarrow$ no. of node inside tree)



Leafs = $\left(\lfloor \frac{n}{2} \rfloor + 1 \text{ to } n\right)$ — (follow in any leaf)

$= \left(\frac{6}{2} + 1 \text{ to } 6\right)$

$= (4 \text{ to } 6)$



leafs = $\left(\lfloor \frac{n}{2} \rfloor + 1 \text{ to } n\right)$

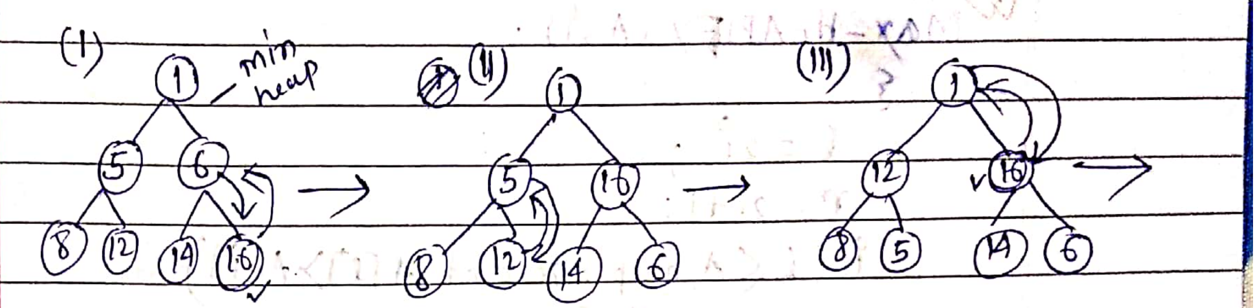
$= \left(\lfloor \frac{8}{2} \rfloor + 1 \text{ to } n\right)$

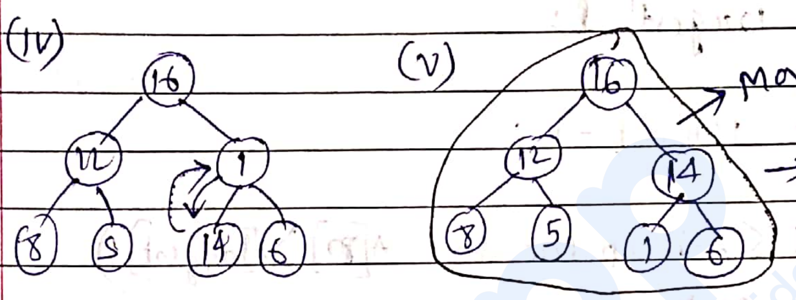
$= 5 \text{ to } 8$

MAXHE

ex:

1	5	6	8	12	14	16

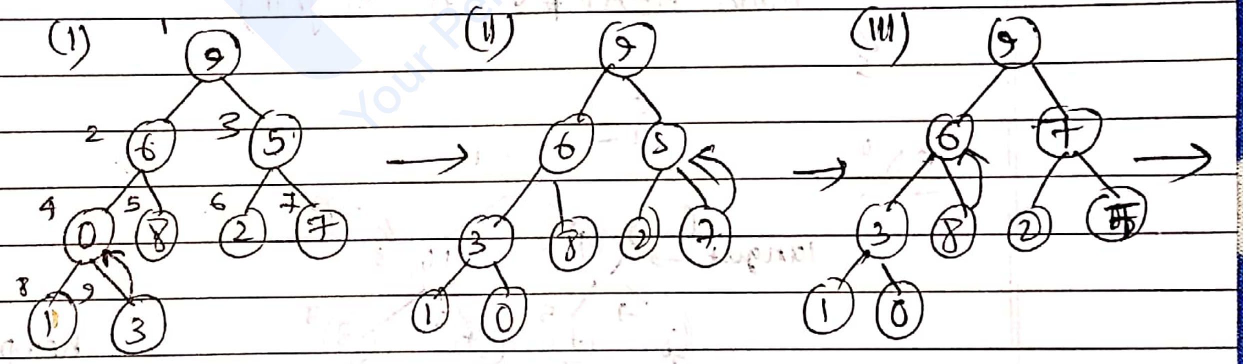




16	12	14	8	5	1	6

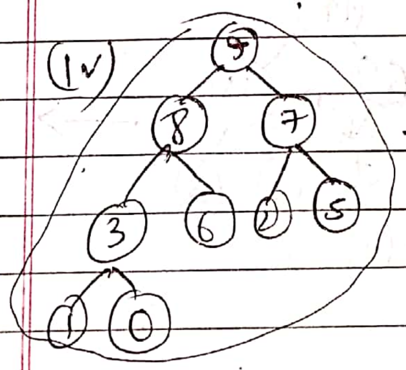
ex:

1	2	3	4	5	6	7	8	9
9	6	5	0	8	2	7	1	3



leaf $= \lfloor \frac{n}{2} \rfloor + 1$ to n

$= (5 \text{ to } 9)$



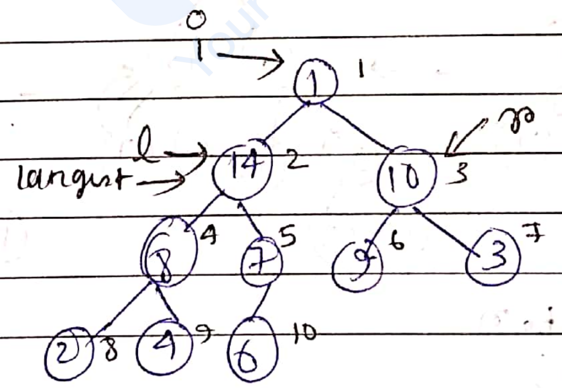
→ following Max heap Properties.

→ Every leaf is a Max heap.

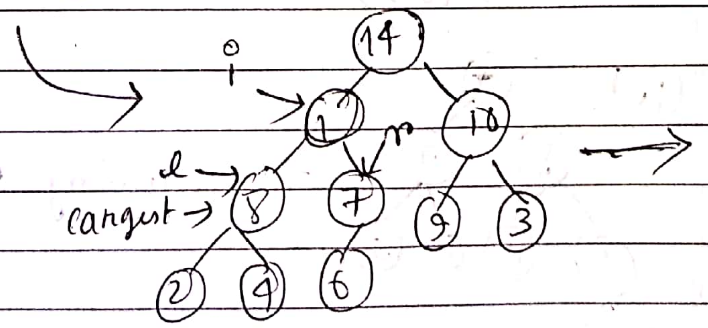
(Max-heapify algorithm)

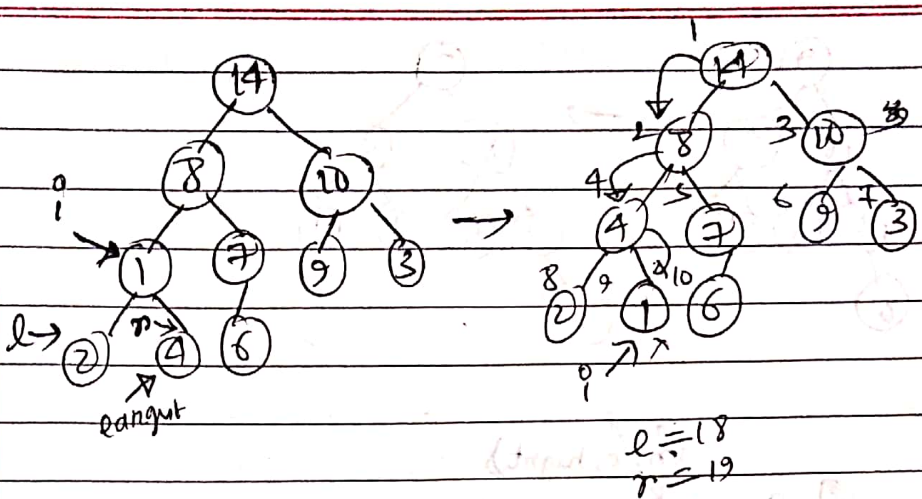
```
MAX-HEAPIFY (A, i).
{
   l = 2i;
   r = 2i+1;
   if (l ≤ A.heap size and A[l] > A[i])
        largest = l;

   else largest = i;

   if (l ≤ A.heap size and A[r] > A[largest])
      largest = r;

   if (largest ≠ i)
      exchange A[i] with A[largest]

      MAX-HEAPIFY (A, largest)
}
```

ex:



heap size = 10





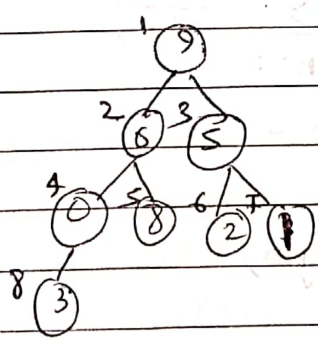
Total time complexity, $(2 \times \log n) = \boxed{O(\log n)}$

Space complexity, no. of level
$= \boxed{O(\log n)}$

• (Build max heap algorithm)

```
BUILD-MAX-HEAP(A)
{
    A.heapsize = A.length
    for (i = ⌊A.length/2⌋ downto 1):
        MAX-HEAPIFY(A,i)
}
```

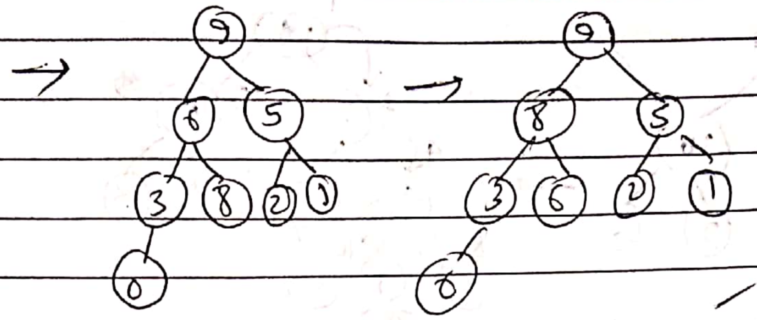
$\boxed{ex}$: 9, 6, 5, 0, 8, 2, 13, 3

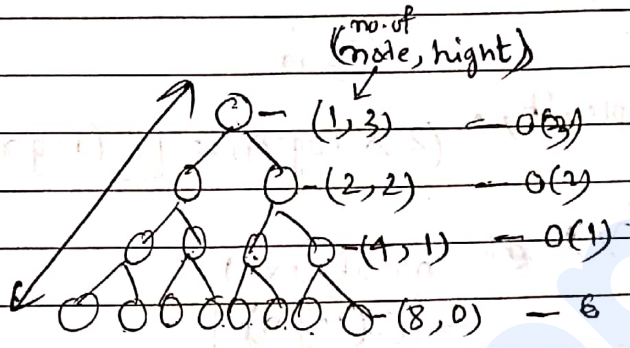


$(1 \text{ to } \lfloor n/2 \rfloor)$ – non leaf

$(\lfloor n/2 \rfloor + 1 \text{ to } n)$ – leaf.

$n \to 8$





Maximum no. of node present in ~~level~~ hight "h" $= \boxed{\left\lceil \frac{n}{2^{h+1}} \right\rceil}$

$(h \to 0)$ $= \left\lceil \frac{15}{2^{0+1}} \right\rceil = 8$

$(h \to 1)$ $= \left\lceil \frac{15}{2^{2}} \right\rceil = 4$

total time $= \sum_{h=0}^{\log n} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h)$ or ch

$= \frac{cn}{2} \sum_{h=0}^{\log n} \left(\frac{h}{2^h} \right)$

$= O\left(\frac{cn}{2} \sum_{h=0}^{\infty} \frac{h}{2^h} \right)$ $\to 2$

In order build a max heap –

time complexity $= O(n)$

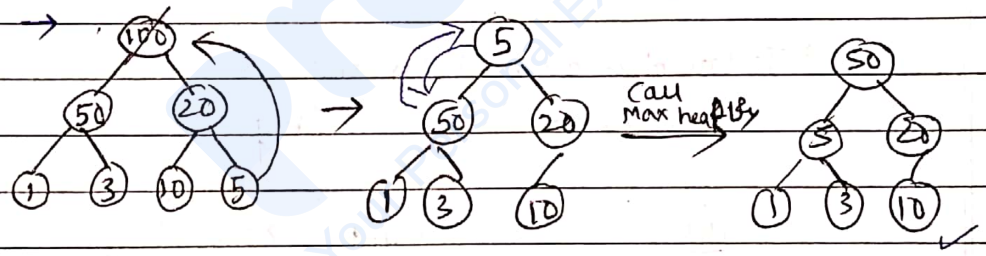
space complexity $= O(\log n)$

- Extract-max from ~~m~~ MAX HEAP :

HEAP-EXTRACT-MAX (A)
{
  if (A.heap-size < 1)
    error "heap underflow"
  max = A[1]
  A[1] = A[A.heap-size]
  A.heap_size = A.heap_size − 1
  (Constant time)
  MAX-HEAPIFY (A, 1) } O(log n) time.
  return max;
}

ex:

100, 50, 20, 1, 3, 10, 5 , Delete-max value (root value)

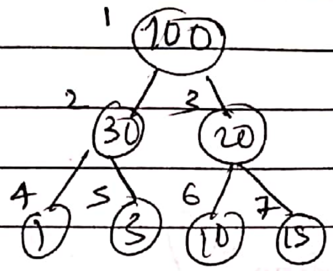


— Total time complexity = O(log n)
Space complexity = O(log n).

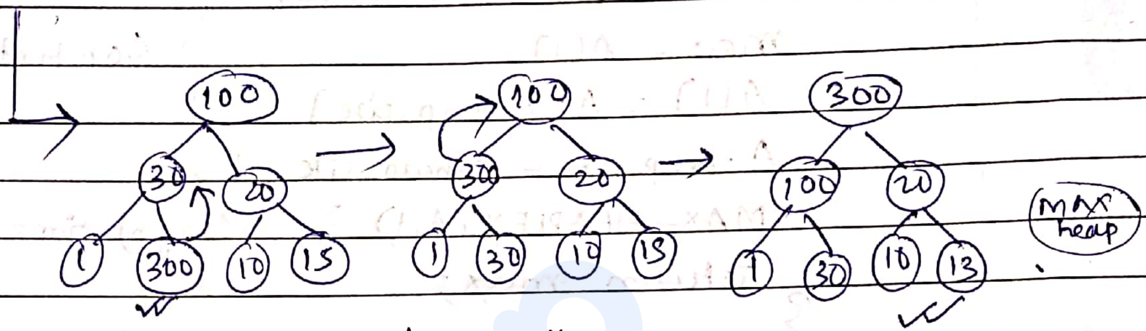
- HEAP-Increase Key (Max-heap)

HEAP-increase-key (A, i, key)   i → Index number
{
  if (key < A[i])
    error
  A[i] = key
  while (i > 1 and A[i/2] < A[i])
  Change A[i] and A[i/2] ; ~~i~~ i = i/2 ; }

$\boxed{ex}$: Increase element at index 5 by 300 (key)



$\begin{cases} i=5 \\ Key = 300 \end{cases}$

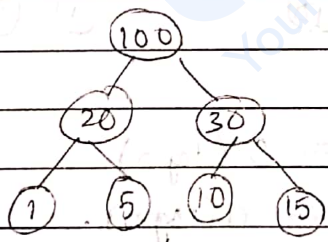


to Increase or decrease key —

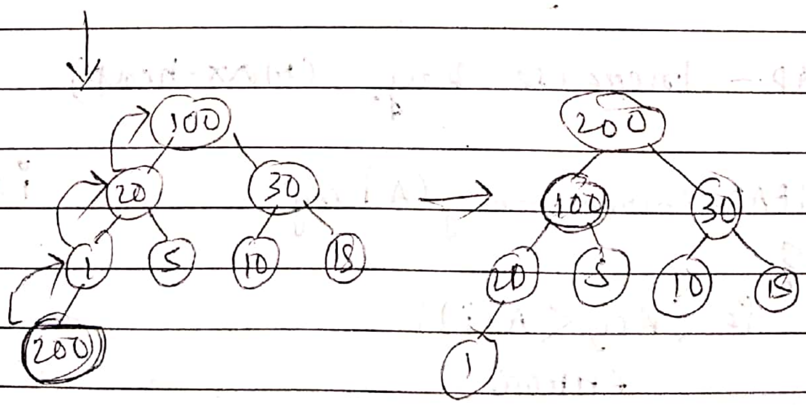
→ Time Complexity = $O(\log n)$.

- Insert key into max-heap =

→ To Insert a element in max heap, Time Complexity is = $O(\log n)$.



Insert - 200



Date
Page

heap-operation

Max heap

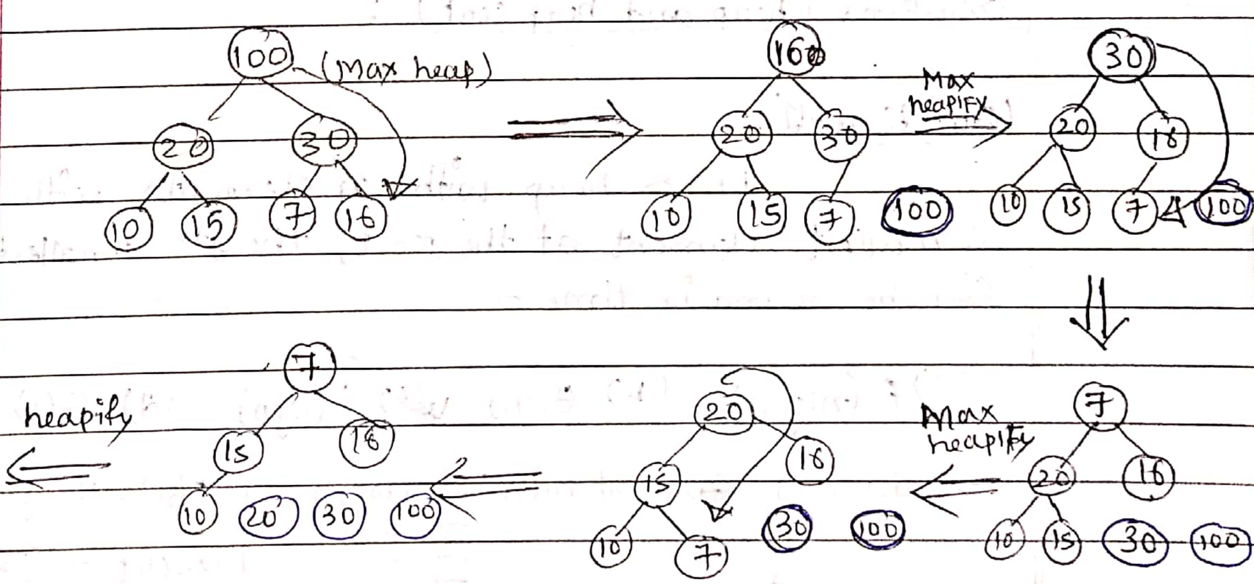
find max	Delete max	insert	Key increase	Key decrease
$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$

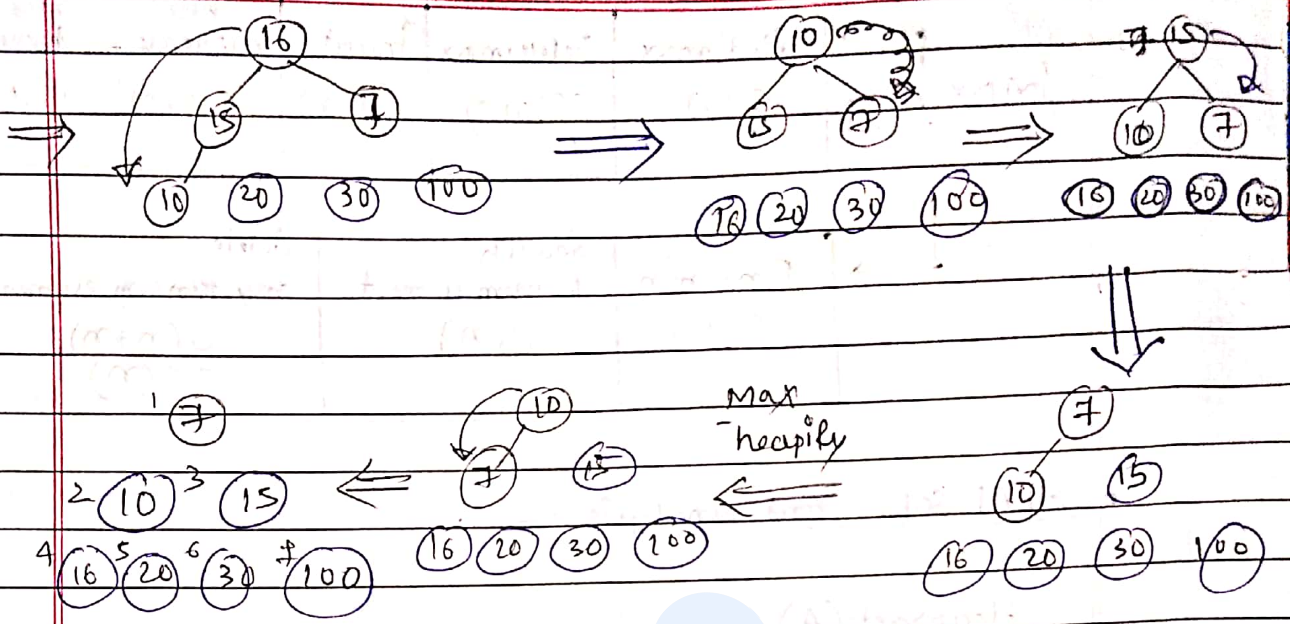
Find min	Search random element	delete any random element
$O(n)$	$O(n)$	$O(n+n) = O(n)$

- HEAP SORT and analysis =

```
HeapSort(A)
{
  BUILD-MAX-HEAP(A)
  for( i = A.length downto 2 )
  exchange A[1] with A[i]
  A.heap size = A.heapsize - 1;
  Max-HEAPIFY(A,1)
}
```

→ | 100 | 20 | 30 | 10 | 15 | 7 | 16 |





1	2	3	4	5	6	7
7	10	15	16	20	30	100

(array sorted)

→ $\boxed{\text{Time complexity (Heap Sort)} = O(n \log n)}$

T(log n) → for hight of ~~tree~~ heap. ~~learning hight of tree~~

T(n) → for heapify.

Questions (heap and heap sort) :

Question - 1

In a heap with 'n' elements with the smallest element at the root, the 7th smallest element can be found in time -

(a) θ (nlogn) (b) θ(n) (c) θ (logn) (d) θ(1)

→ delete 1st ~~element~~ min — O(logn)
" 2 — "
" 3 — "
" 4 — "
" 5 — "
" 6th min — O(logn)

Find 7th = O(1)

Insert = 6 * O(logn)

Time = ~~6(logn~~ 6 * O(logn) ~~+~~ + 6 * O(logn) + O(1)
= O(logn)

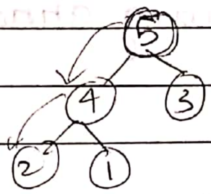
Question - 2

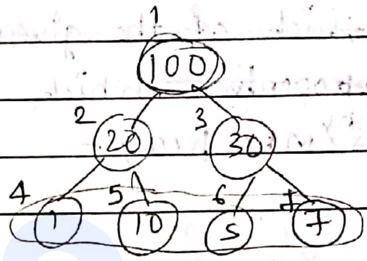
In a binary max heap containing 'n' numbers the smallest element can be found in time -

(a) $\Theta(n)$ (b) $\Theta(\log n)$ (c) $\Theta(\log\log n)$ (d) $\Theta(1)$.

→

5	4	3	2	1





- smallest element

$O(\log n) + O(1)$

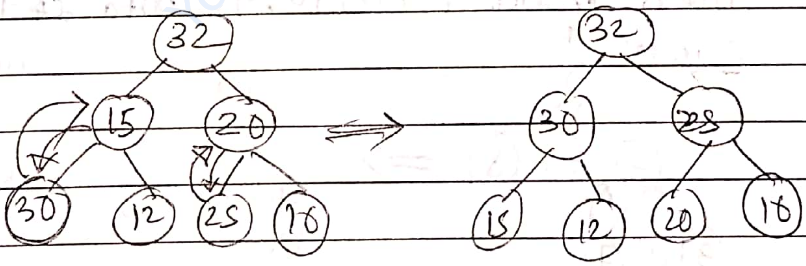
$leaf = (\lfloor n/2 \rfloor + 1 \text{ to } n)$
$= (4 \text{ to } 7)$

Time taken to find ~~mini~~ smallest no. $= O(n)$.

Question - 3

32, 15, 20, 30, 12, 25, 16 this element Inserted into a Max heap what is resulting heap look like -

→



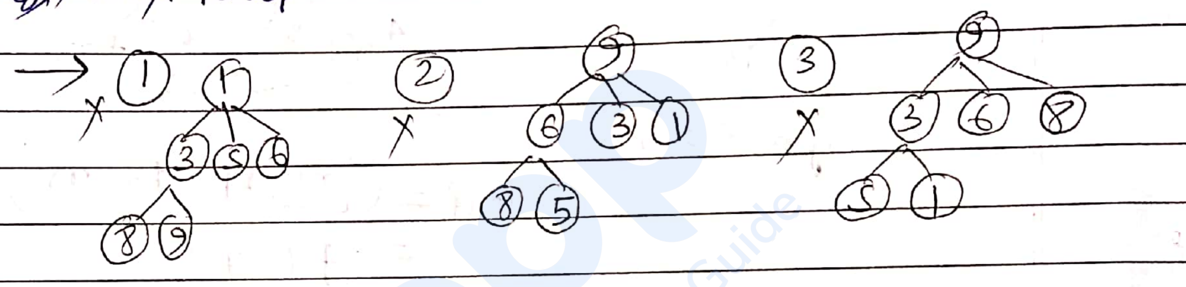
resulting order (32, 30, 25, 15, 12, 20, 16)

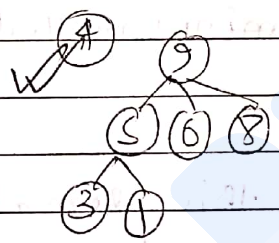
Question - 4

3-ary Max heap,

1. 1, 3, 5, 6, 8, 9
2. 9, 6, 3, 1, 8, 5
3. 9, 3, 6, 8, 5, 1
4. 9, 5, 6, 8, 3, 1

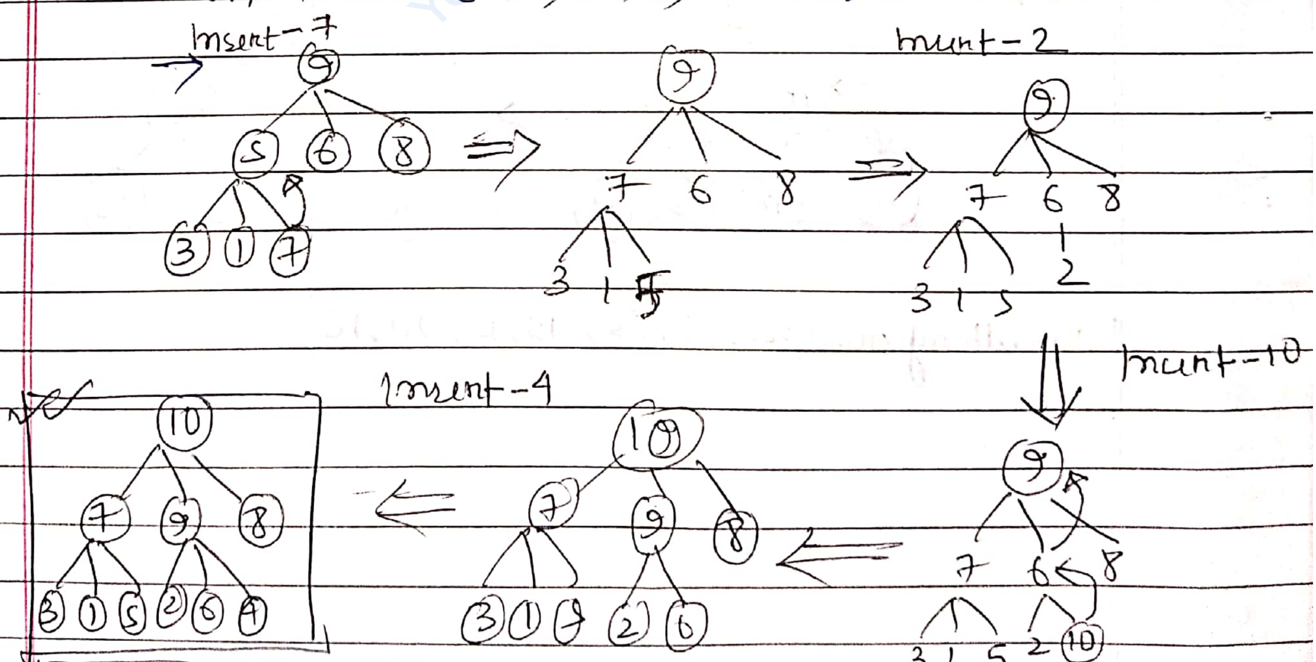
~~Which of the given sequence follow 3-ary Max heap property~~ which of the following array represent 3-ary max heap -





(always parent is greater than children)

then insert (7, 2, 10, 4) into ~~st~~ the result -

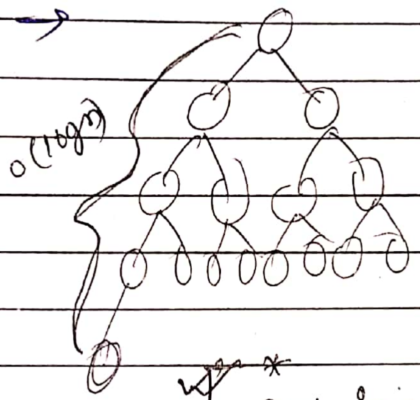


10, 7, 9, 8, 3, 1, 5, 2, 6, 4

Question - 5

Consider the process of inserting an element into a max heap. If we perform a binary search on the ~~par~~ path from new leaf to root to find the position of newly inserted element, the number comparisons performed are —

→



heap-hight (log n)
when n element

Binary search $\theta = \log n$

$= \boxed{O(\log \log n)}$

* applying Binary search on some problem which are have Time complexity of O(n) is not going to reduce the complexity to O(log n)

Question - 6

We have a binary heap on 'n' elements and wish to insert 'n' more elements (not necesarily one after another) into this heap. The total time required for this is —

→ 2n
↓ call (BUILD heap)

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

→ put all n element into array and then call build heap.

for 'n' element T.C $= O(n)$
2n " Time.c $= O(2n) = O(n)$.

Date
Page

Greedy Algorithm

www.gatenotes.in

- Introduction of Greedy algorithm —

~~cd~~ Optimization :

given a problem If I try to minimize or maximize there property then it is call optimization problem.

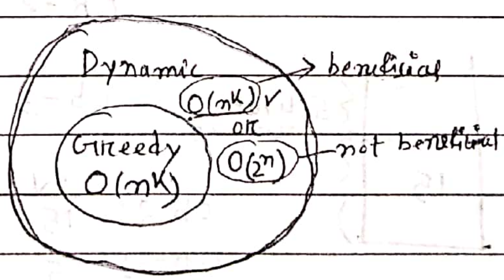
optimization Problems :
- → minimize cost.
- → max profit.
- → maximize reliability.
- → minimize risk.

→ Greedy method and dynamic programming are two programming peradence ~~whith~~ which ~~are~~ could be used to solve optimization problem.

→ Using Greedy method we will not be able to Solve all the optimization problem.

→ ~~Dynam~~ Dynamic program can solve any optimization problem. (We want to apply Dynamic programing if can do ~~ca~~ somthing better ~~then~~ compare to ~~exclutive~~ exsustive search)

→ Time Complexity of dynamic programming could be — $O(n^k)$ or $O(2^n)$



~~Greedy Knapsack algorithm~~

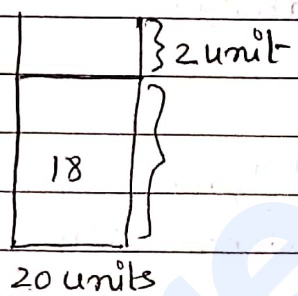
- **Knapsack problem :**
  ↳ means bag

	Object-1	ob-2	ob-3
profit	25	24	15
weight	18	15	10

~~W=2~~
M = 20 (M - capacity of bag)
n = 3 (n - no. of object)

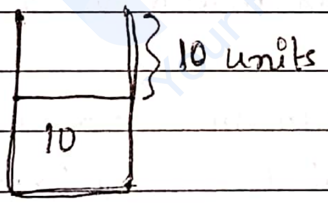
15w − 24p
2w = 24 * 2/15

→ when Greedy about profit −



	Weight	Profit
ob-1 :	18	25
ob-2 :	2	(24)(2/15)
	20	28.2

→ when Greedy about weight −



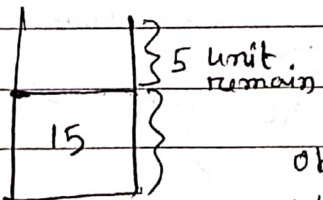
	W	P
ob-3 :	10	15
ob-2 :	10	(24) * (10/15)
	20	31

→ When Greedy about ratio of profit & weight.

profit per units , ob-1 : $\frac{25}{18}$ = 1.4

ob-2 : $\frac{24}{15}$ = 1.6

ob-3 : $\frac{15}{10}$ = 1.5



	W	Profit
ob-2 :	15	24
ob-3 :	5	(15)(5/10)
	20	31.5

→ when Greedy about ratio of prof. more profit ~~compare to~~

Date
Page

Algorithm =

Greedy Knapsack
{

for $i = 1$ to n;

Compute P_i / w_i; — $O(n)$

sort objects in non increasing order of P/w. — sorting algo take (merge sort) or heap sort $O(n \log n)$

for $i = 1$ to n from sorted list.

if $(m > 0$ && $w_i \le M)$

$M = M - w_i$;

$P = P + P_i$;

} $O(n)$

else

break;

if $(m > 0)$ — $O(1)$

$P = P + P_i \left(\frac{M}{w_i}\right)$;

}

~~ex~~ ex : 1 What is max profix get out of it —

$M = 15$, $n = 7$

$i =$	1	2	3	4	5	6	7
Objects	1	2	3	4	5	6	7
profits	10	5	15	7	6	18	3
Weight	2	3	5	7	1	4	1
(I) $\frac{P_i}{w_i} =$	5	1.6	3	1	6	4.5	3

(II)

5	1	6	3	7	2	4

fraction ←	2
	7
	3
	6
	1
	5

M=15 units.

$M = \cancel{15}\ \cancel{14}\ \cancel{12}\ \cancel{8}\ \cancel{3}\ \cancel{2}\ 0$

$P = 6 + 10 + 18 + 15 + 3 + 5\left(\frac{2}{3}\right)$

$= 55.3$

TO DOWNLOAD THE COMPLETE PDF

CLICK ON THE LINK GIVEN BELOW

WWW.GATENOES.IN

GATE CSE NOTES