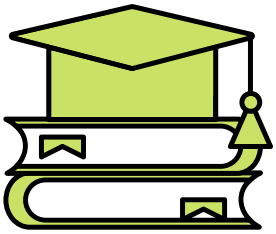


Gatenotes.in

GATE
Computer Science

Ravindrababu Ravula GATE CSE

Hand Written Notes

-: SUBJECT :-

Algorithms

NAME: Rakesh Nama STD.: ____ SEC.: ____ ROLL NO.: ____ SUB.: Algorithm.

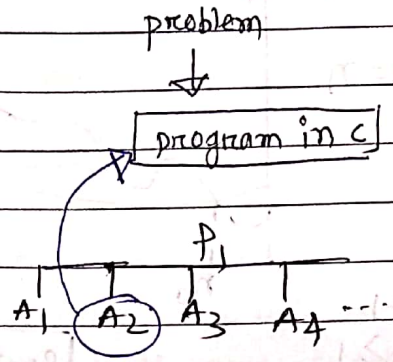
S. No.	Date	Title	Page No.	Teacher's Sign / Remarks
1.		Time and space Analysis.		
2.		sorting Techniques.		
3.		Greedy Algorithms.		
4.		Dijkstra Algorithms.		
5.		Bellman-ford algorithm.		
6.		Shortest paths in DAGs.		
7.		Dynamic programming. + Matrix chain - Multiplication.		
8.		longest common subsequence.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

Time and space analysis

www.gatenotes.in

atlantis
Date
Page

- Introduction to asymptotic notations =

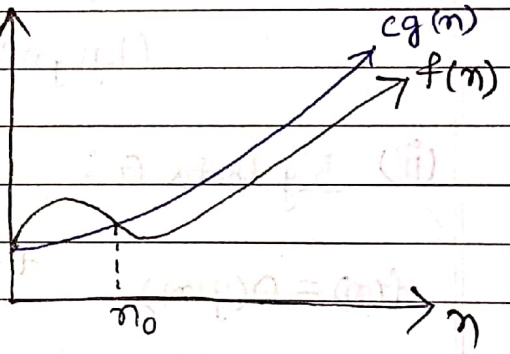


analysis of an algorithm by —
(i) Time (less time)
(ii) Memory (less memory)

(i) Big (Oh) O :

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

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



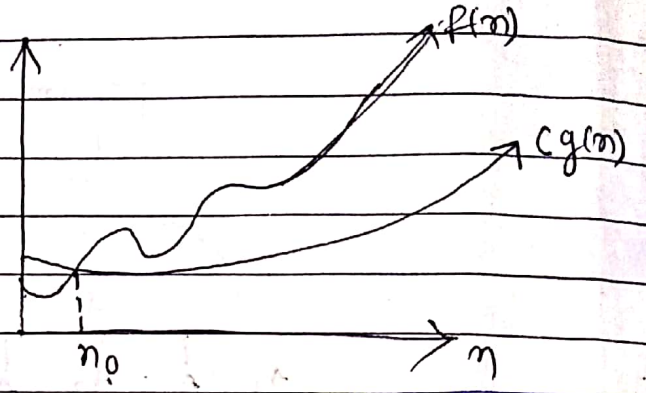
$3n+2 \leq cn$.
$c = 4$
$3n+2 \leq 4n$
$\Rightarrow n \geq 2$

$f(n) \leq cg(n)$
$n \geq n_0$
$c>0,\ n_0 \geq 1.$
$f(n) = O(g(n))$

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

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

(ii) Big Omega Ω :



ex:

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

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

$f(n) \geq cg(n)$.

$3n+2 \geq cn$ $\quad c=1$

$n_0 \geq 1$

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

$f(n) \geq cg(n)$, $n \geq n_0$

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

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

$\downarrow$

$\log n$

$\downarrow$

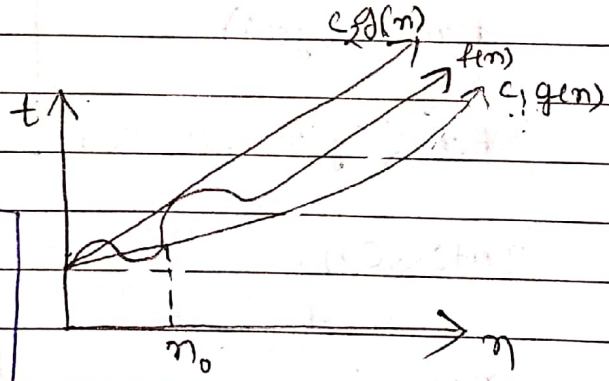
$(\log\log n)$

lowers

take always closest ↑ bound

here closest bound is 'n'.

(iii) Big theta Θ :



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

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

$c_1, c_2 > 0$

$n \geq n_0$

$n_0 \geq 1$

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

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

→ Interested.

O	Ω	Θ
↓	↓	↓
Worst case time are the Upper bound.	Best case	Average case

→ Average case (Θ) 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 = $\Omega(1)$ - (x found out 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

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

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

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

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

→ ~~Any~~ A 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).

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()
{ p=1;                    i <= √n
  for (i=1; i*i <= 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

$k=n*100$

$100 + 2*100 + 3*100 + 4*100 + 5*100 + \cdots + n*100$

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

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

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

atlantis
Date
Page

⑥

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

  for(i=1; i<=n; i++)
  {
    for(j=1; j<= i^2; j++)
    {
      for (k=1; k= n/2; k++)
      { pf
          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= n² times k= n/2 * n²

→ $n/2 + n/2*4 + n/2*9 + n/2*16 + n/2*25 + \cdots + n/2*n^2$

→ $n/2\ (1+2^2+3^2+4^2+\cdots+n^2)$

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

→ $\frac{1}{12} n(n^2+n)(2n+1)$

→ $1/12\, n(2n^3+n^2+2n^2+n)$

→ $1/12\, n(2n^3+3n^2+n)$ → Time Complexity = $O(n^3)$.

$f(n) = n^k + n^{k-1}, \cdots = O(n^k)$

atlantis
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, ... n
2^0 2^1 2^2 2^3, ... 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++)  — Indepent loop — n/2
    for(j=1; j<=n/2; j++)  — n/2
      for(K=1; K<=n; K=K*2)  — log2 n
        pf("ravi");
}
```

→ $n/2 * n/2 * \log_2 n$

→ $\frac{n^2}{4} \log_2 n \rightarrow O(n^2 \log_2 n)$ ← 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");
}
```

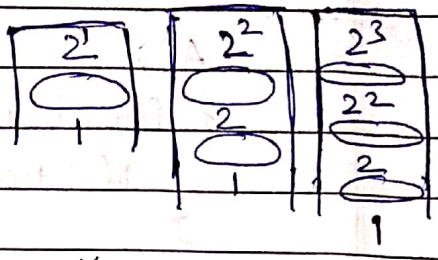
→ $n/2 (\log_2 n)^2 \rightarrow O(n (\log_2 n)^2)$ ← Time complexity
(removed constants)

(10)

assume $n \geq 2$

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

3	$\log_3 n$
5	$\log_5 n$
i	



$n = 2^K$

$K = \log_2 n$

Time complexity $= O(\log_2 n)$.

(11)

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

$i=1$	$i=2$	$i=3$	$i=K$
$j=1$ to n	$j=1$ 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)

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

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

$\rightarrow n \log n$

$\rightarrow$ Time complexity $= O(n \log n)$.

(12)

```
A()
{ int n = 2^K;
  for (i=1; i<=n; i++)          ---- (n)
  {
    j=2
    while (j<=n)                 (2^8)
    {
      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

→ $n*(K+1)$

→ $n(\log\log n + 1)$

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

$n = 2^{2^K}$

$\log_2 n = 2^K$

$K = \log_2 \log_2 n$

atlantis
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 recursion program

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

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) =

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

$\boxed{T(n) = O(n)}$ → Time complexity.

(2)

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

$= 1$; $n = 1$ 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)

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

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

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

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

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

$= n + (n-1) + (n-2) + \cdots + 2 + 1$ → A.P

$= \frac{n(n+1)}{2} = \frac{n^2+n}{2}$ (here most significant term n^2)

So, time complexity - $O(n^2)$.

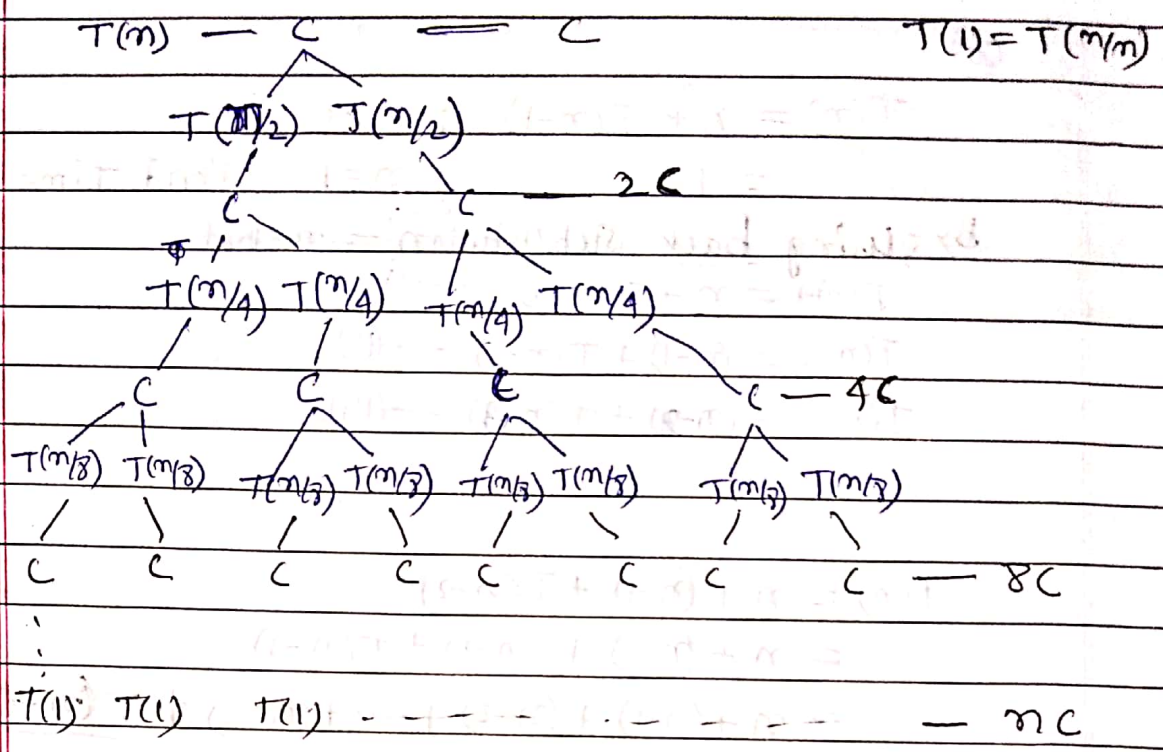
(3) Recursion tree method

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

$= C$ $n = 1$

Find time complexity using Recursion tree method -

→ $T(n) = 2T(n/2) + C$.



$\rightarrow C + 2C + 4C + 8C + \cdots + nC$

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

let $n = 2^k$

$\rightarrow C(2^0 + 2^1 + 2^2 + 2^3 + \cdots + 2^k)$ — GP

$\rightarrow C \frac{(2^{k+1} - 1)}{(2-1)}$

$\rightarrow C(2^{k+1} - 1)$

$\rightarrow C(2^k \cdot 2 - 1)$

$\rightarrow C(2n - 1)$

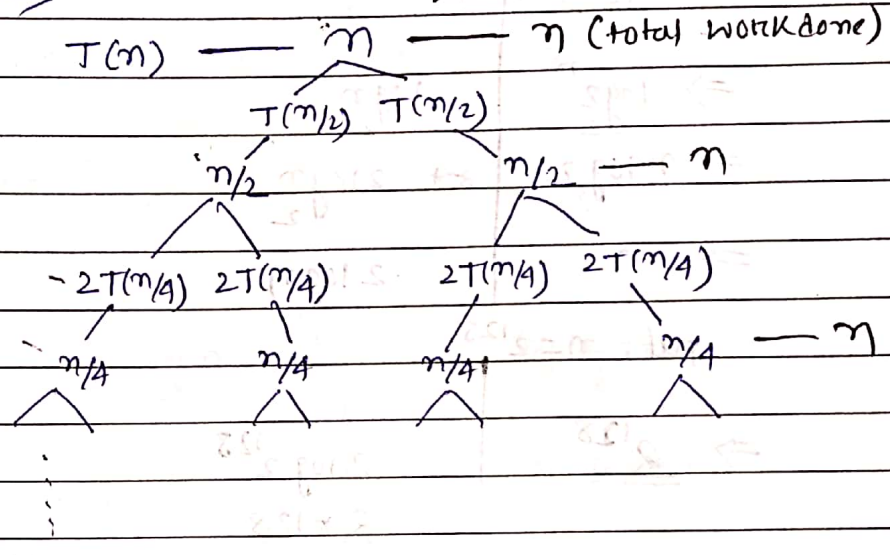
$\Rightarrow O(n)$ → Time complexity.

④ $T(n) = 2T(n/2) + n \; ; \; n \geq 1$

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

find time complexity using Recursion tree method=

→ Work done each level –



→ $n/2^0 \to n/2^1 \to n/2^2 \to n/2^3 \to \cdots \to \left(\frac{n}{2^k}\right)^1$ $\boxed{n = 2^k}$

$k = \log_2 n$

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

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

→ $n(k+1)$

→ $n(\log_2 n + 1)$

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

5 ~~Easiest way to solve recursion~~ = ~~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	$n \log n$
$\rightarrow n \times n$	$n \log n$
$\rightarrow n$	$\log n$

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

ex-3

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

n is larger than $\log n$.

ex-4 $n^{\log n} > n \log n$

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

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

$n = 2^{128}$

$\rightarrow 128 * 128$ | $\rightarrow 128 + 7$

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

ex-5 $\sqrt{\log n} > \log\log n$

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

$n = 2^{10}$

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

$= 5$ | $= 3.5$

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

ex-6 $n^{\sqrt{n}} > n^{\log n}$

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

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

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

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

$n = 2^{128}$

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

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

$g(n) > f(n)$

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

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

$f_1 > f_4 > f_2 > f_3$

→ This is how functions are Compared.

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

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

atlantis
Date
Page

- **Masters theorem** = (to find time complexity of recursion program easily)

$$T(n) = aT(n/b) + \theta(n^k \log^p n)$$

$a \geq 1$, $b > 1$, $k \geq 0$ 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 $T(n) = O(n^k)$.

Ex-1

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

→ after compare with Masters equation —

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

$$a \quad < \quad b^k$$
$$\| \qquad \| $$
$$3 \quad < \quad 2^2$$

iii) a)

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

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

Ex-2

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

→ After compare with masters equation,

$a=4,\ b=2,\ K=2,\ P=0$

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

$a = b^K$

ii) → a)

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

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

Ex-3

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

→ After compare with masters Equation -

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

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

$a < b^K$

iii) a) $T(n) = \theta(n^K \log^P n)$

$= \theta(n^2)$

Ex-4

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

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

Ex-5

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

→ $a=16, b=4, K=1, P=0$

$a=16 > b^K=4$

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

$= \theta(n^2)$

Ex-6 $T(n) = 2T(n/2) + n\log n$.

→ $a=2, b=2, K=1, P=1$

$a=2 = b^K=2$

$a=b^K$

ii) a)

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

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

Ex-7

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

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

→ $a=2, b=2, K=1, P=-1$

$a=b^K$

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

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

atlantis
Date
Page

Ex-8

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

→ $a=2, b=4, K=0.51, P=0$

$a=2 < b^K = 4^{0.5}$

iii) a) $T(n) = \theta(n^K \log^P n)$

$T(n) = \theta(n^{0.51})$

Ex-9

$T(n) = 0.5T(n/2) + 1/n$ ✗

→ $a=0.5, b=2, K=-1, P=0$ (not possible using masters theorem).

$a=0.5$ $b^K = 2^{-1} = 1/2 = 0.5$

$a = b^K$ a should be $a \geq 1$

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

$= \theta(n^{\log_2 0.5} \log n)$

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

→ here, $a=6, b=3, K=2, P=1$

$a < b^K$

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

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

→ $a=7,\ b=3,\ k=2,\ p=0$

$a < b^k$

∴ i) a) $T(n) = \theta(n^2)$

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

→ $a=4,\ b=2,\ k=0,\ p=1$

here, $a > b^k$

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, $a > b^k$

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

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

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

Ex-15

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

→ $a=2,\ b=2,\ k=1/2,\ p=0$

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

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

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

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

Ex-16

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

→ $a=3,\ b=2,\ k=1,\ p=0$

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

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

Ex-17

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

→ $a=3,\ b=3,\ k=1/2,\ p=0$

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

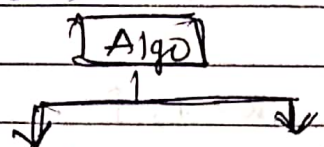
iii) a)

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

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

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

Analysis space complexity of iterative and recursive Algorithm =



- **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]; ✓   → Space complexity = O(n)
  for(i=1 to n)
    B[i]= A[i];
  }
```

(3)
```
Algo(A,1,n)
{ creat B[n,n] —— n^2
  int i,j; —— 2          (n^2+2)
  for(i=1 to n)
    for(j=1 to n)   → here space complexity = O(n^2).
  { B[i,j] = A[i]
```

atlantis
Date
Page

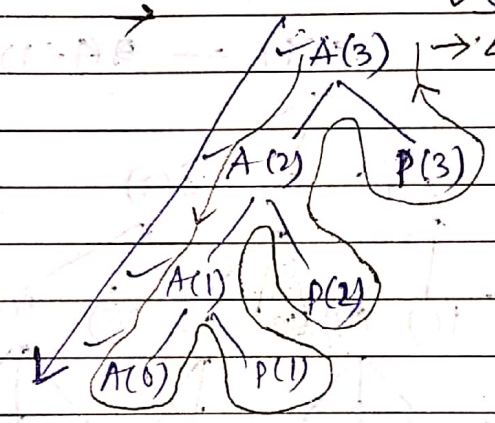
• Space complexity of recursive (Algorithm) program =

→ When ~~the recursive 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);
}
```

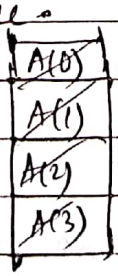
✓ Space complexity = O(n)



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 -

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

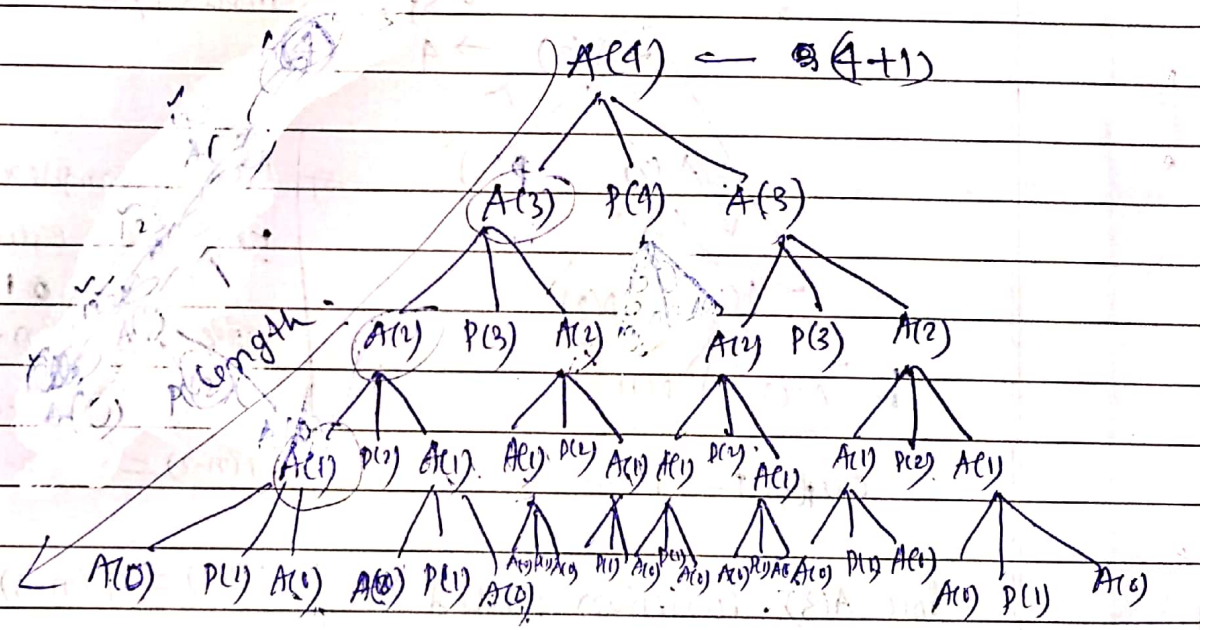
$T(n-1) = T(n-2)+1$
$T(n-2) = T(n-3)+1$

$T(n) = T(n-3)+3$
⋮
$T(n) = T(n-K)+K$
$= T(n-n)+n$
$= T(0)+n$
$= 1+n$
$T(n) = 1+n$
$= O(n)$.

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)
= 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 ramule function called n+1 times)

Time complexity =

Recursive equation =

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

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

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

$n - K = 0$
$K = n$

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

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

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

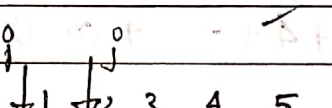
Sorting Techniques

- **Insertion sort algorithm and analysis =**

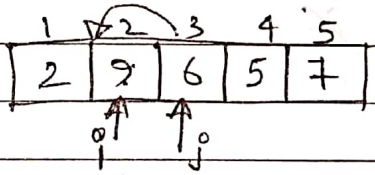
```
Insertion-sort (A)
{
  for(j=2 to A.length)
  {
    Key = A[j];   // insert A[j] into stored sequence
                     A[1....j-1]
    i = j-1
    while (i>0 and A[i] > Key)
      { A[i+1] = A[i].
        i = i-1 }
    A[i+1] = Key
  }
```

Working Procedure =

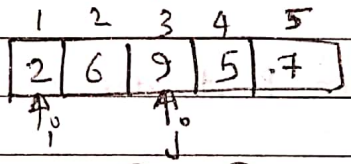
→ Array



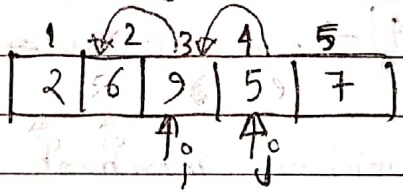
(1)



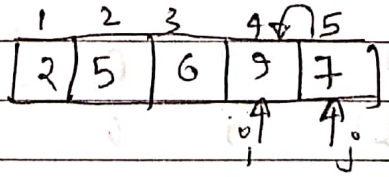
(2)



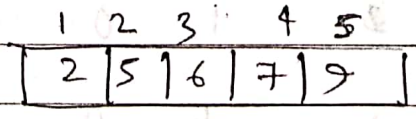
(3)



(4)



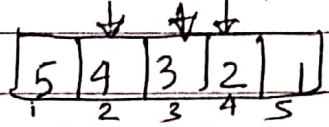
(5)



(Array sorted)

atlantis
Date
Page

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



→ 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)

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

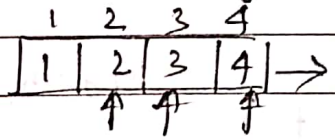
$= 2\left(1+2+3+4+\cdots+(n-1)\right)$ (A.P) $\quad \left[\frac{n(n+1)}{2}\right]$

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

$= n^2 - n$

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

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



→ when array ~~is~~ 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(n²) (T.C)
double linked list	O(n)	O(1)	= O(n) (T.C)

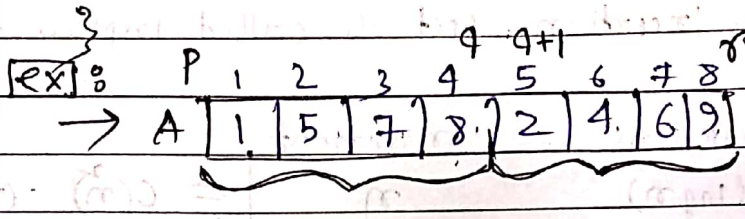
- Merge sort algorithm and analysis =

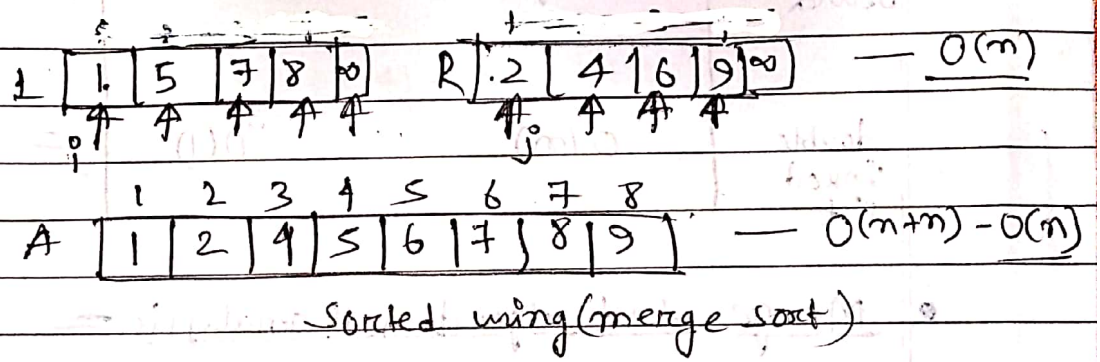
```
MERGE (A, P, q, r)   [Merge procedure]
{
  n1 = q - p + 1;
  n2 = r - q;
  Let L [1 ... n1+1] and R [1 to n2+1] be new arrays
  for (i = 1 to n1)
     L[i] = A[p+i-1]

  for (j = 1 to n2)
     R[j] = A[q+j];

  L[n1+1) = ∞;
  R[n2+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;
```





Sorted using (merge sort)

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

⇒ ~~Merge procigure is also called~~
~~Merge procigure is also called out of place proci~~
→ Merge procedure is also called out of place procedure.

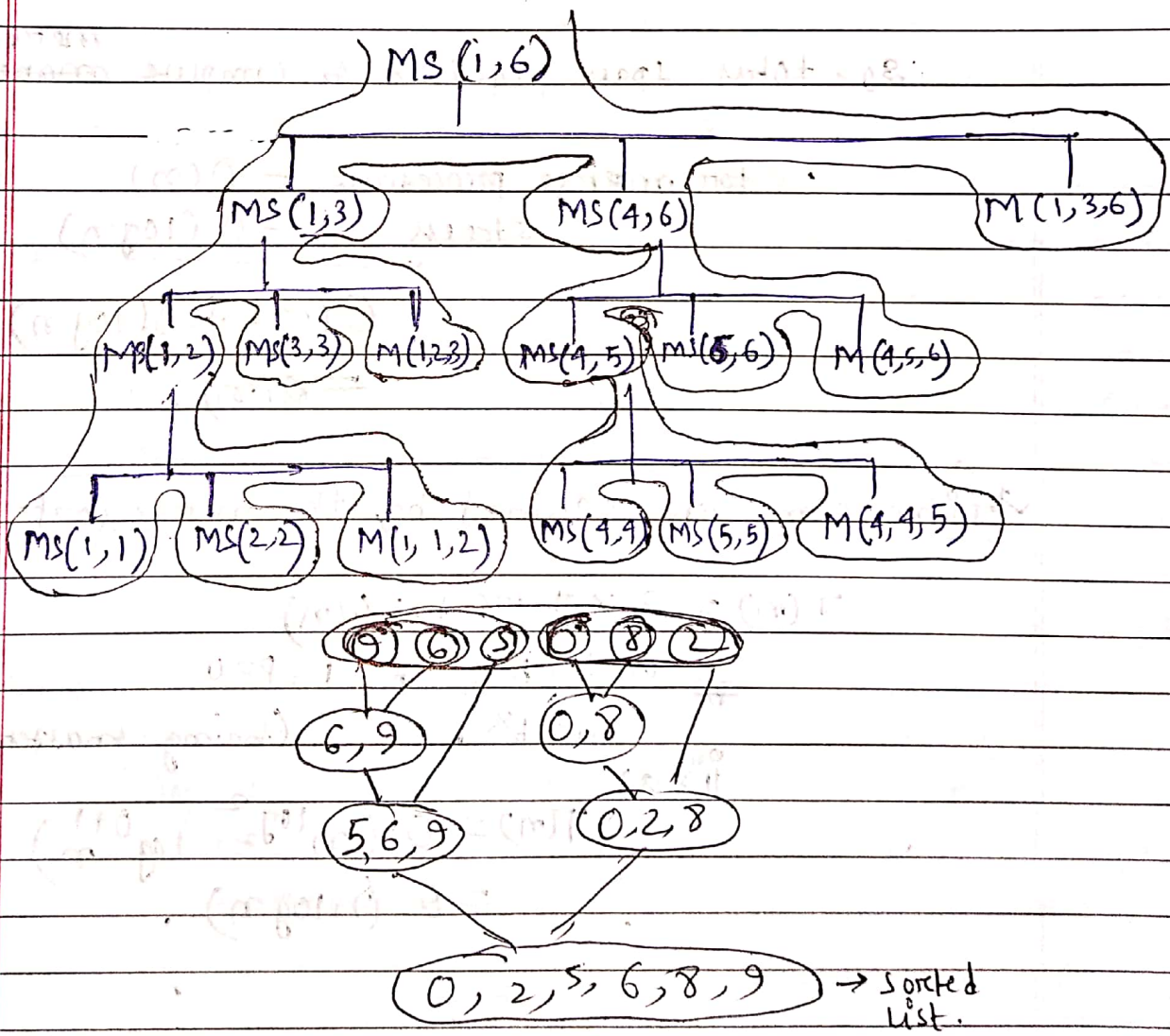
[Merge-sort]

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:

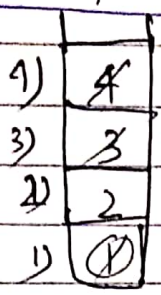
	1	2	3	4	5	6
A	9	6	5	0	8	2

P r



atlantis
Date
Page

Space Complexity required by the merge sort - O(n)



6 — 4 level

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

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

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

So, total space required to compute 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 — $O(n \log n)$

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

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

$a = b^K$ (Using masters theorem)

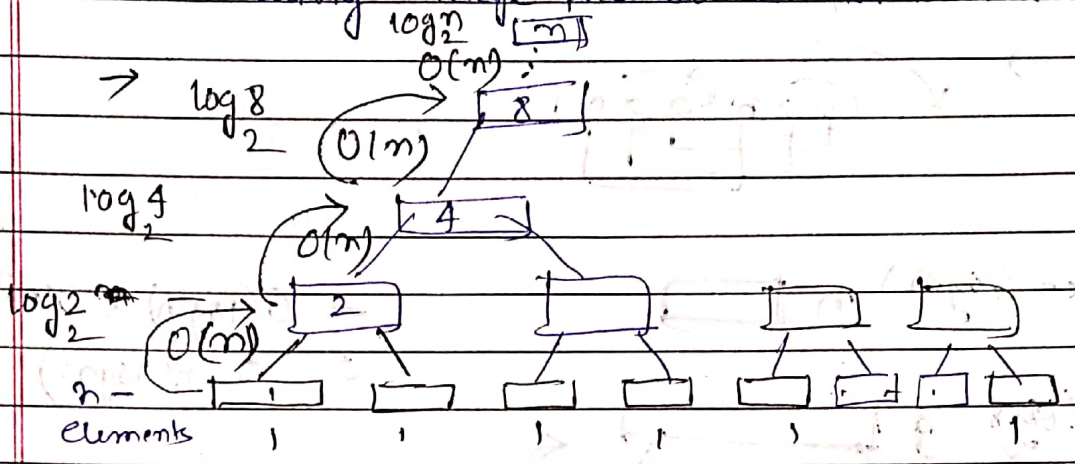
$a) \; T(n) = \theta(n^{\log_2 2} \log^{0+1} n)$

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

atlantis
Date
Page

Q-1 (2-Way merging)

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



Total time = $O(n) * O(\log n)$

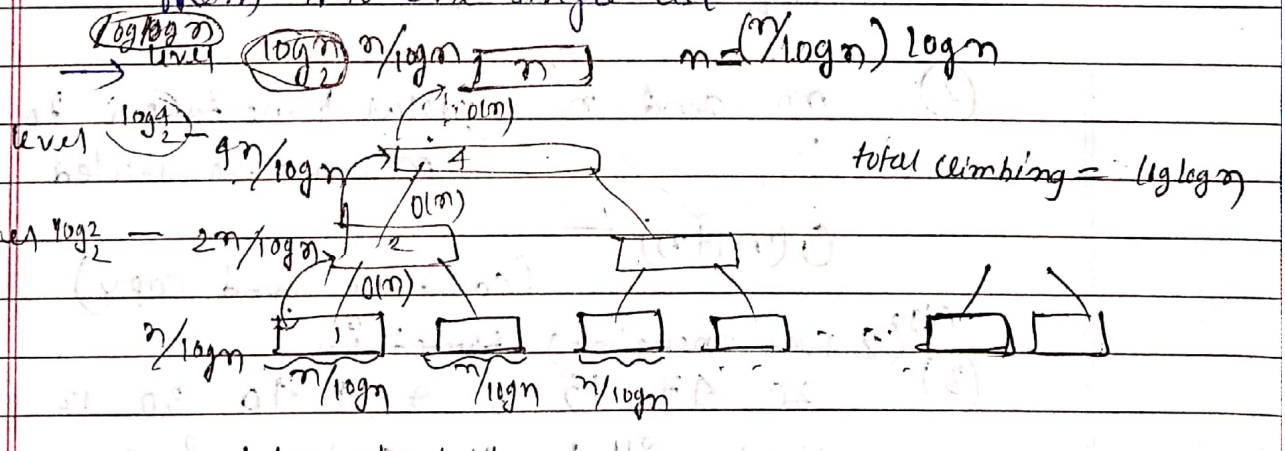
$= O(n \log n)$

$O(n)$ → work done each level.

no. of levels → $O(\log n)$

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 -

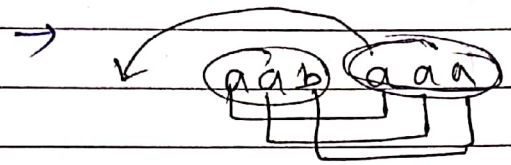


$\log n * n/\log n = n$ element each level.

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

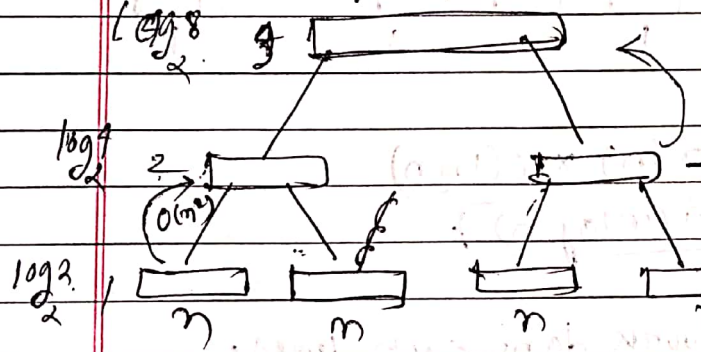
Q-3

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



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

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



$O(n) \times O(n) + O(n^2)$

$= O(n^2)$

$= O(n^2)$ eliminate

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 —

$O(m+n)$

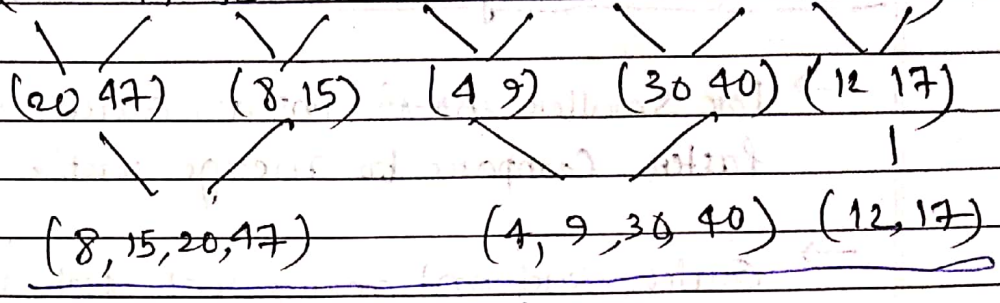
(Compare and copy)

(3) Algo (2 way merging) ~~sorted in~~

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)



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

atlantis
Date
Page

• Quick Sort Algorithm =

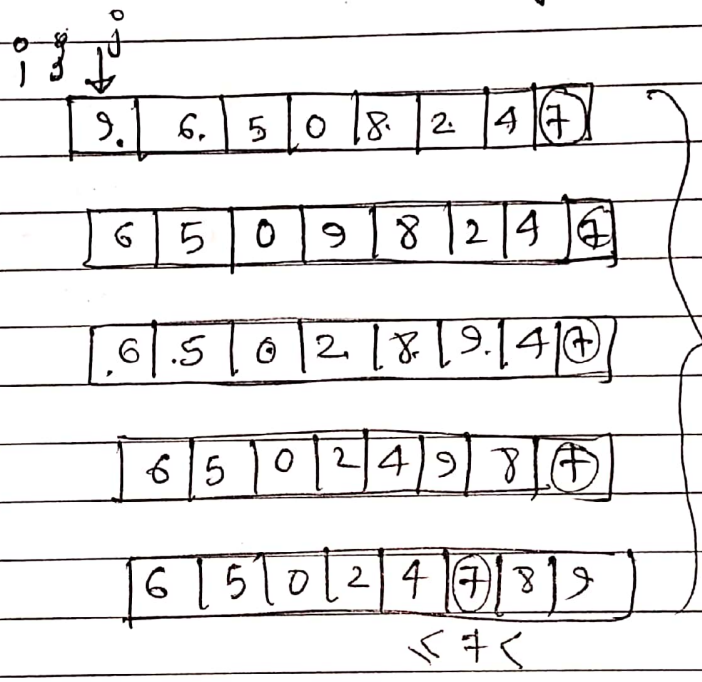
→ For smaller ~~no. of~~ input, quick sort return faster compare to merge sort.

→ Quick sort and merge sort ~~both~~ follow divide and conquer method.

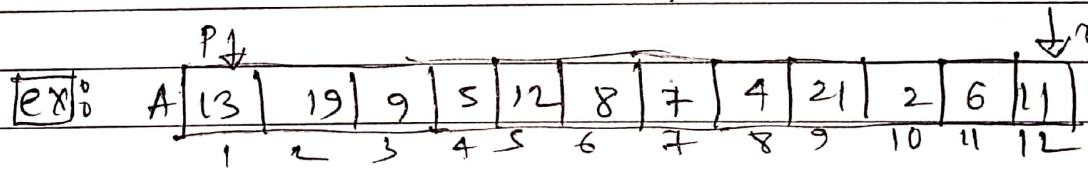
• Partition Algorithm =

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

ex:



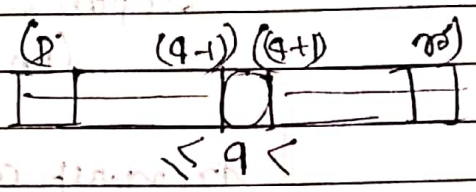
ex:




```
PARTITION (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]
}
}
exchange A[i+1] with A[r]
return i+1;
}
```

QUICKSORT (A, p, r) — T(n)

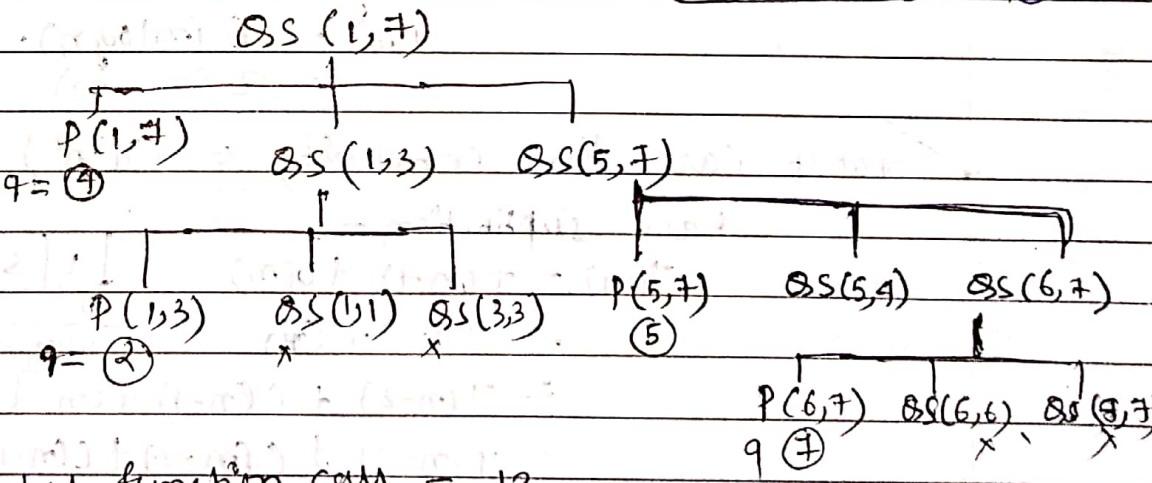



```
{
    if (p<r)
    {
        q = PARTITION (A,p,r); — O(n)
        QUICKSORT (A, p, q-1); — T(n/2)
        QUICKSORT (A, p, q+1); — T(n/2)
    }
}
```

ex:

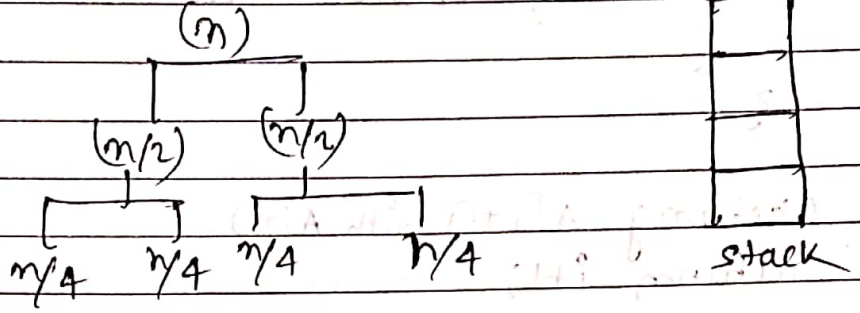
	1	2	3	4	5	6	7
A	5	7	6	1	3	2	4

1	2	3	4	5	6	7



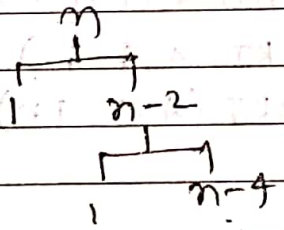
total function call = 13

→ 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 = Θ(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 ≤ $O(n^2)$

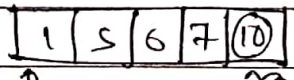
back suphitution –

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

$= T(n-1) + Cn$

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

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



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

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

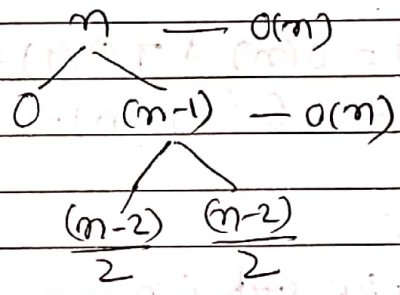
$= O(n^2)$ $= \frac{n(n+1)}{2}$

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

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

ex:

→ best and worst combination -



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

$\leq 2O(n) + T(n/2)$

$= \theta(n \log n)$

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 $= O(n)$

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

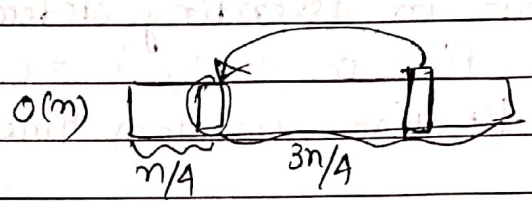
$= O(n) + 2T(n/2)$

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

Question ①

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.

→



partition

$$T(n) = O(n) + O(1) + O(n) + T(n/4) + T(3n/4).$$

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

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

$\Theta(n \log n)$

Question ③

Using Quick sort On an algorithm, given I/p

$1, 2, 3, \cdots n$ — Time taken T_1

$n, n-1, n-2, \cdots 1$ — " " T_2

What is the relationship between T_1 & T_2.

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

$(T_1 = T_2)$.

atlantis
Date
Page

[Question] - ④

partition algo which take O(n) time, we are spliting the problem into two part. (1/5 n), (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)

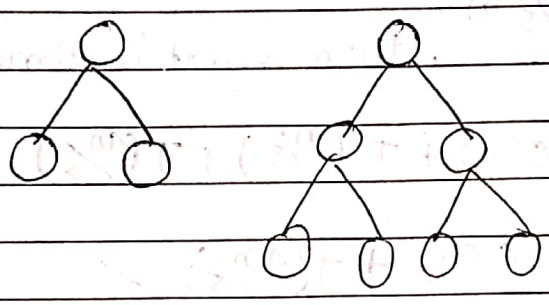
→ heap is ~~a~~ datastructure which used optimise (Time complexity) some of the operation

→ heap
- min heap
- max heap.

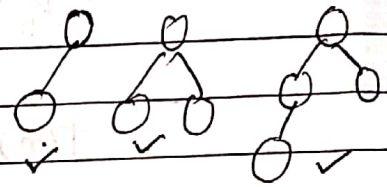
→ using heap we can implement heapsort algorithm.

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

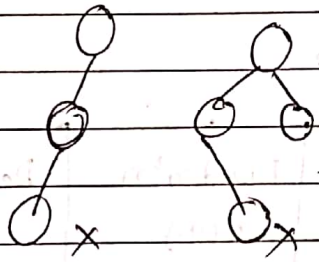
→ Every heap is an almost complete binary tree.



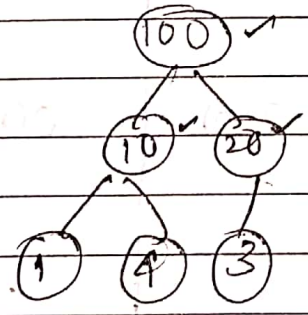
Complete B.T.



almost complete B.T.

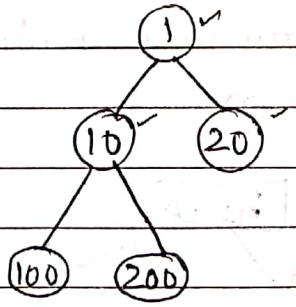


Max-heap :



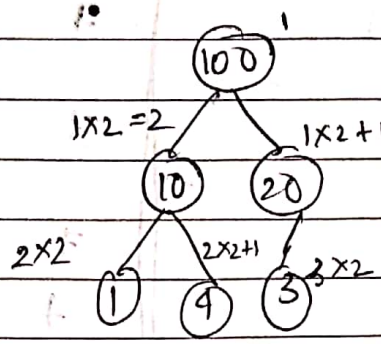
→ all the elements in the root should be grater than leaf.

Min-heap :



→ minimum element will present in the root.

max heap = Store complete binary tree in a array =

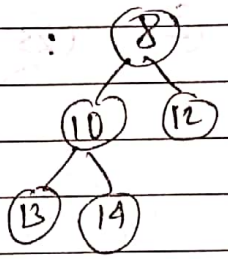


	1	2	3	4	5	6
array	100	10	20	1	4	3

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.

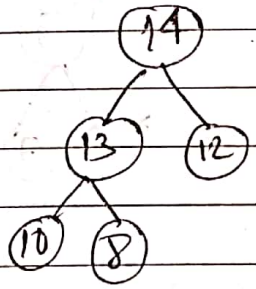
8	10	12	13	14



(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 always greater than its child)

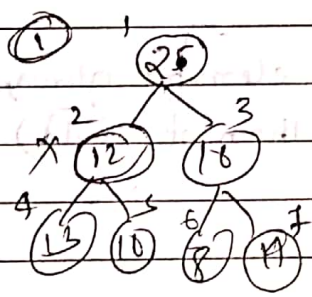
atlantis

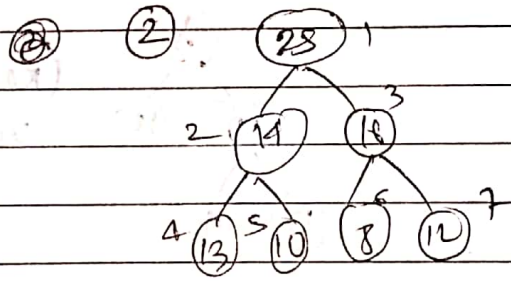
Date

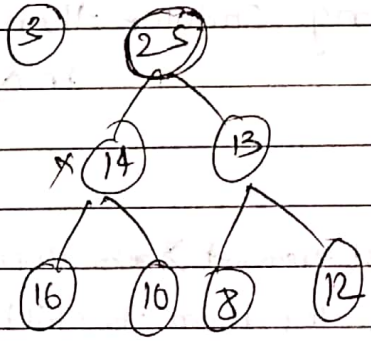
Page

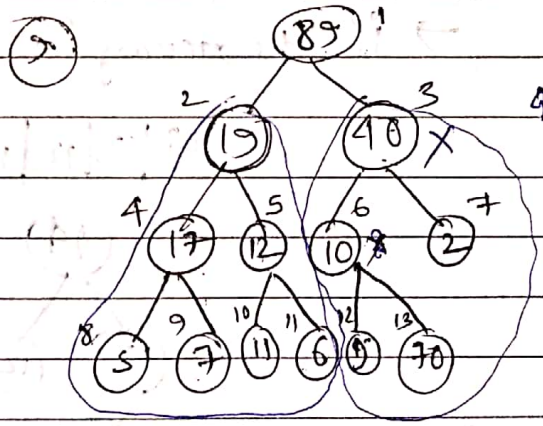
property follow heap ↑

	1 2 3 4 5 6 7	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	1 2 3 4 5 6 7 8 9 10 11 12 13 / 8, 9, 19, 40, 17, 12, 10, 2, 5, 7, 11, 6, 9, 70	13	2



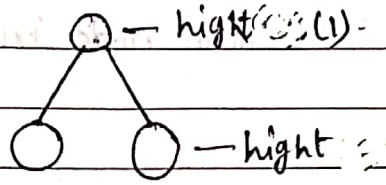






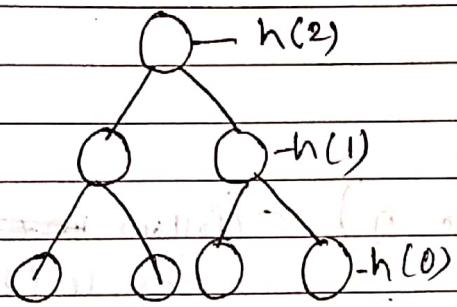
40 < 70 not follow Max heap property

Some properties of complete binary tree =

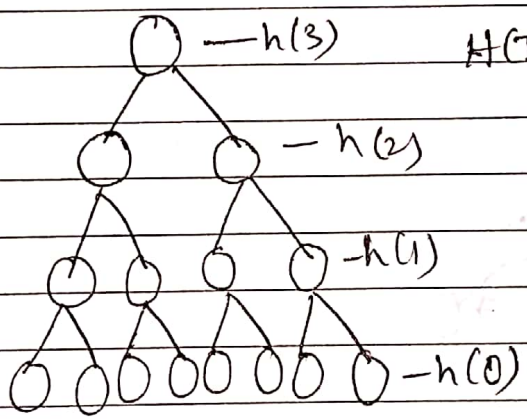


hight of a tree = hight of root.

$H(T) = 1$



$H(T) = 2$



$H(T) = 3$

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

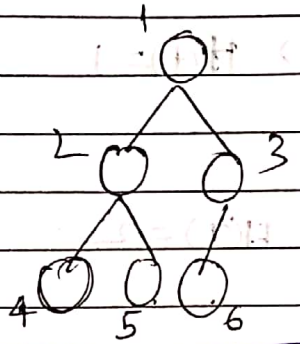
- max no. of node in Complete binary tree = $(2^{h+1}-1)$

($h \rightarrow$ 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 = $\lfloor \log n \rfloor$

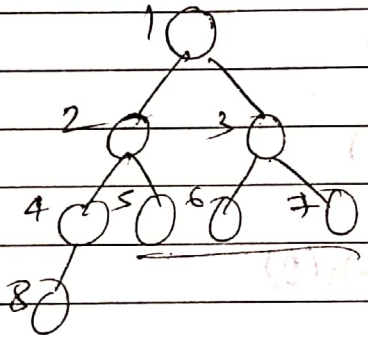
($n \rightarrow$ no. of node inside ~~the~~ tree)



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

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

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



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

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

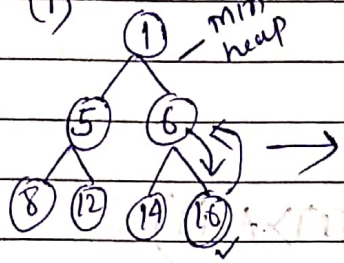
$= 5 \text{ to } 8$

MAX HEAP

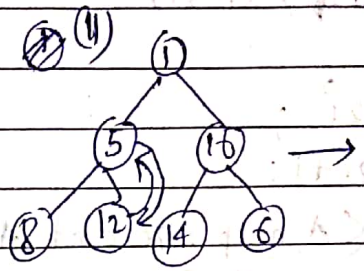
ex:

1	5	6	8	12	14	16

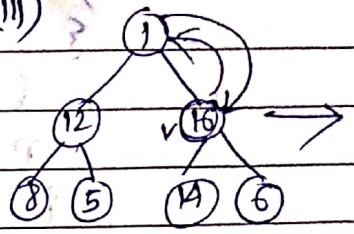
(I)



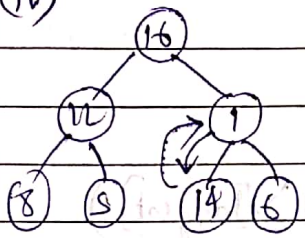
(II)



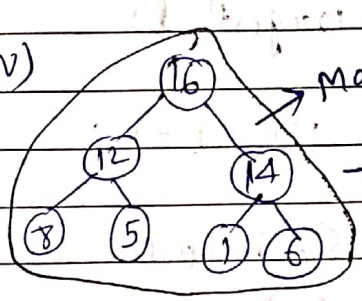
(III)



(IV)



(V)



max heap

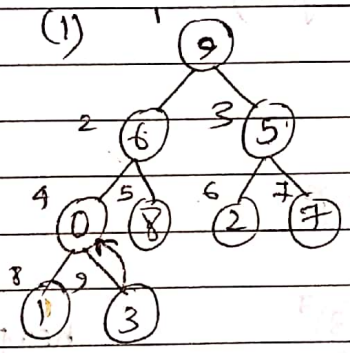
→ following max heap properties.

16	12	14	8	5	1	6

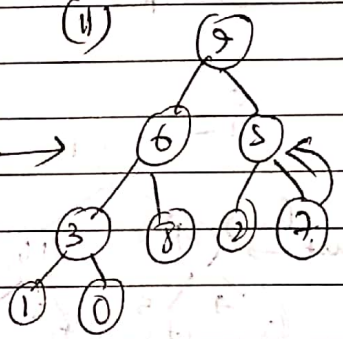
ex:

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

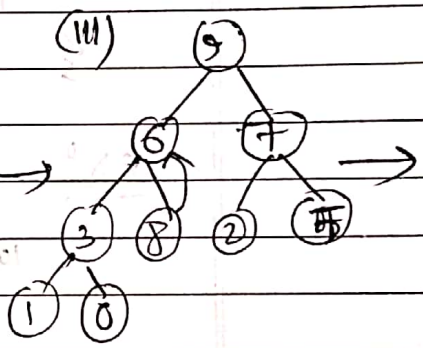
(I)



(II)



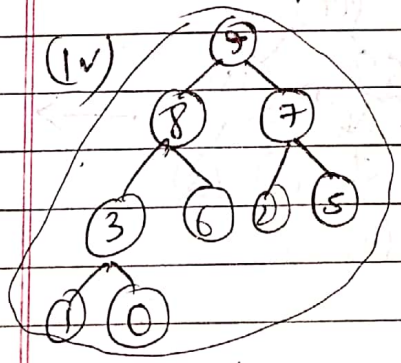
(III)



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

= (5 to 9)

(IV)



→ following max heap properties.

→ Every leaf is a Max heap.

(Max-heapify algorithm)

MAX-HEAPIFY (A, i).

{

$l = 2i$;

$r = 2i + 1$;

if ($l \leq$ A.heap size and $A[l] > A[i]$)

largest = l ;

else largest = i ;

if ($l \leq$ A.heap size and $A[r] > A[largest]$)

largest = r ;

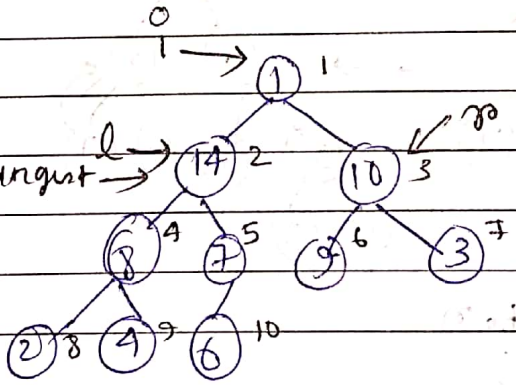
if (largest $\neq i$)

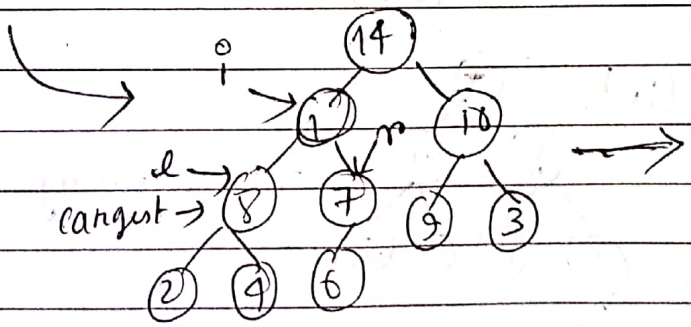
exchange A[i] with A[largest]

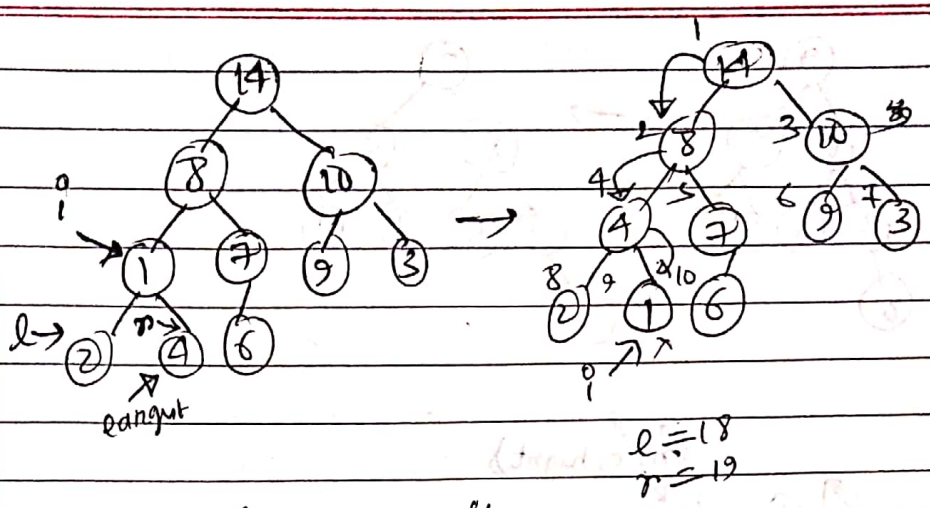
MAX-HEAPIFY (A, largest)

}

ex:







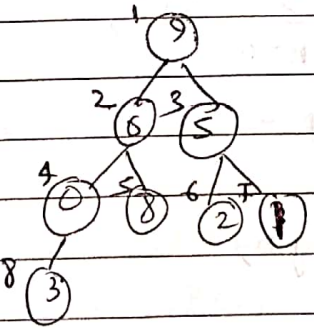
Total time complexity, $(2 \times \log n) = \boxed{O(\log n)}$

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)
}
```

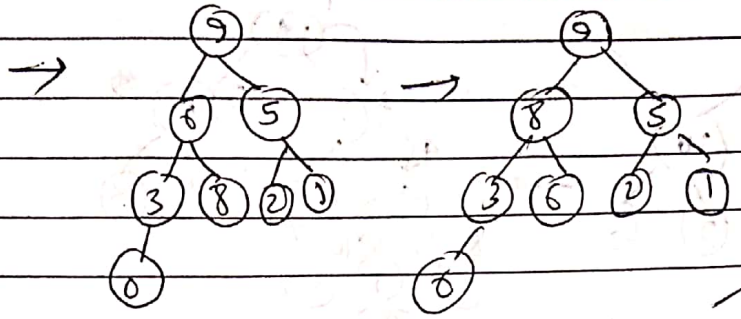
Ex: 9, 6, 5, 0, 8, 2, 13, 3

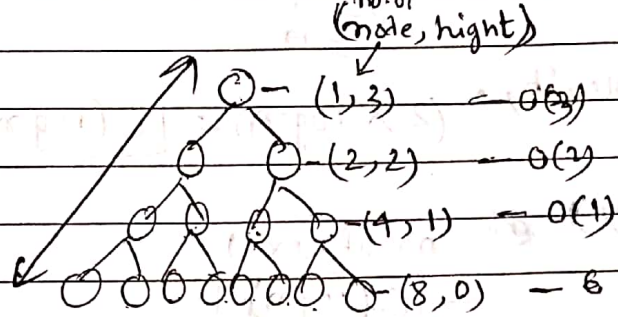


$(1 \text{ to } \lfloor n/2 \rfloor)$ – non leaf

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

$n \to 8$





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

$(h \to 0)$

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

$(h \to 1)$

$= \left\lceil \frac{15}{2^2} \right\rceil = 4$

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

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

$$= O\left(\frac{cn}{2} \left(\sum_{h=0}^{\infty} \frac{h}{2^h} \right) \right) \quad \text{(4 = 2)}$$

In order build a max heap –

Time Complexity $= O(n)$

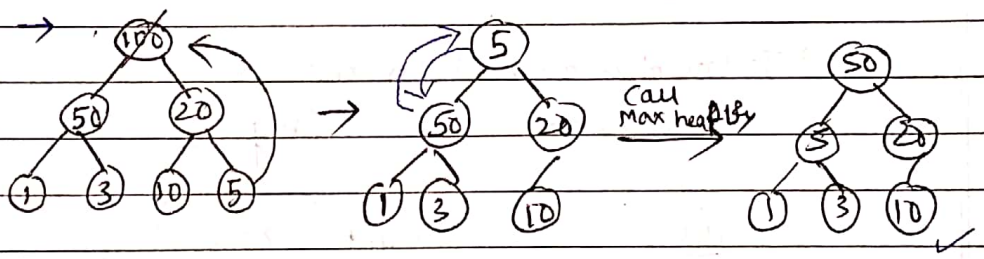
Space complexity $= O(\log n)$

- 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]                   }  Constant time
  A.heap_size = A.heap_size - 1           }
  MAX-HEAPIFY (A,1) } O(logn) time.       }
  return max;
}
```

ex:

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

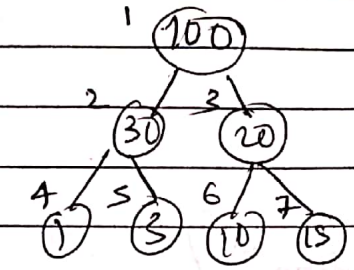


- Total time complexity = O(logn)
  Space complexity = O(logn).

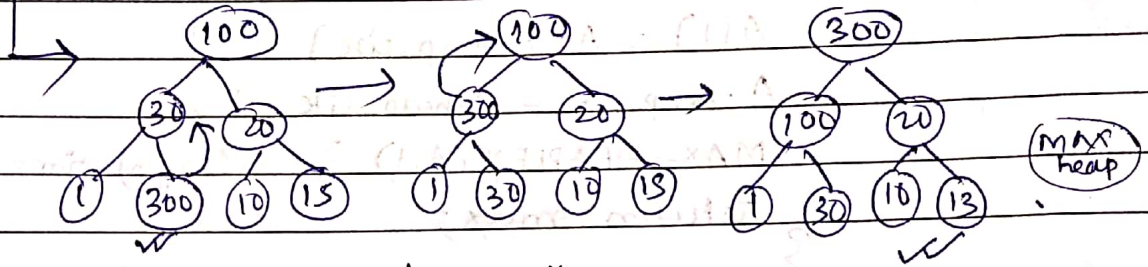
- HEAP - Increase Key (MAX-heap)

```
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/2 ; }
```

[ex]: Increase element at index 5 by 300 (Key)



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

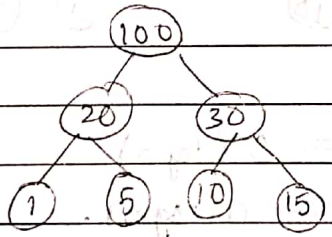


to Increase or decrease Key -

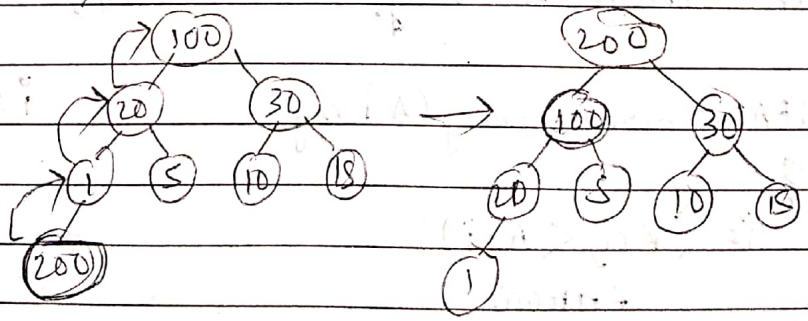
→ - Time Complexity = O(log n).

- **Insert key into max-heap =**

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



Insert - 200



atlantis
Date
Page

(heap-operation)

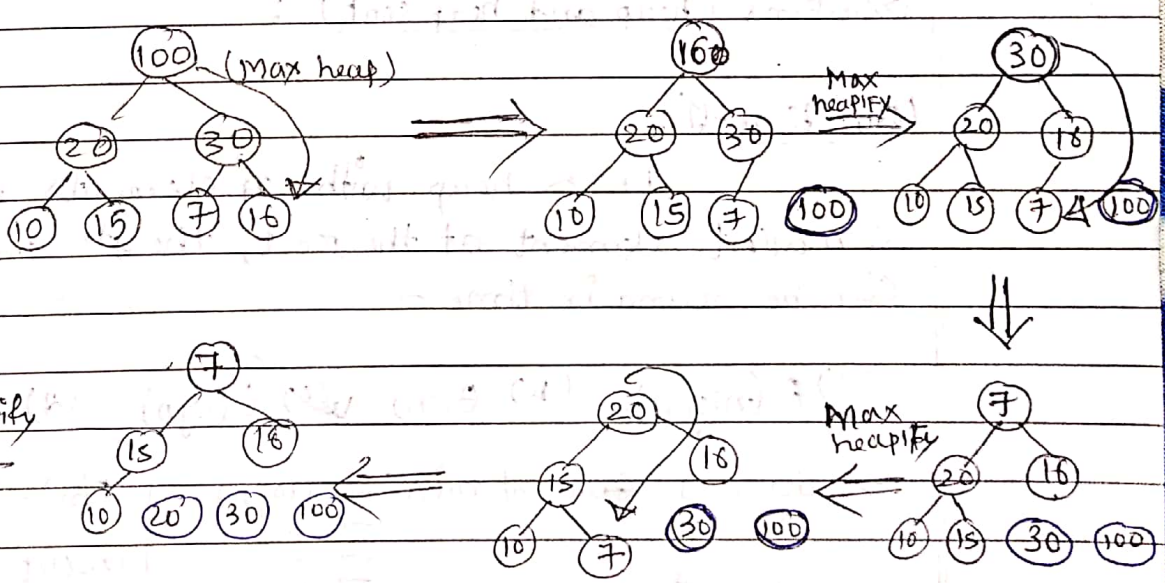
	Find max	Delete max	insert	Key increase	Key decrease
Max heap	O(1)	O(logn)	O(logn)	O(logn)	O(logn)

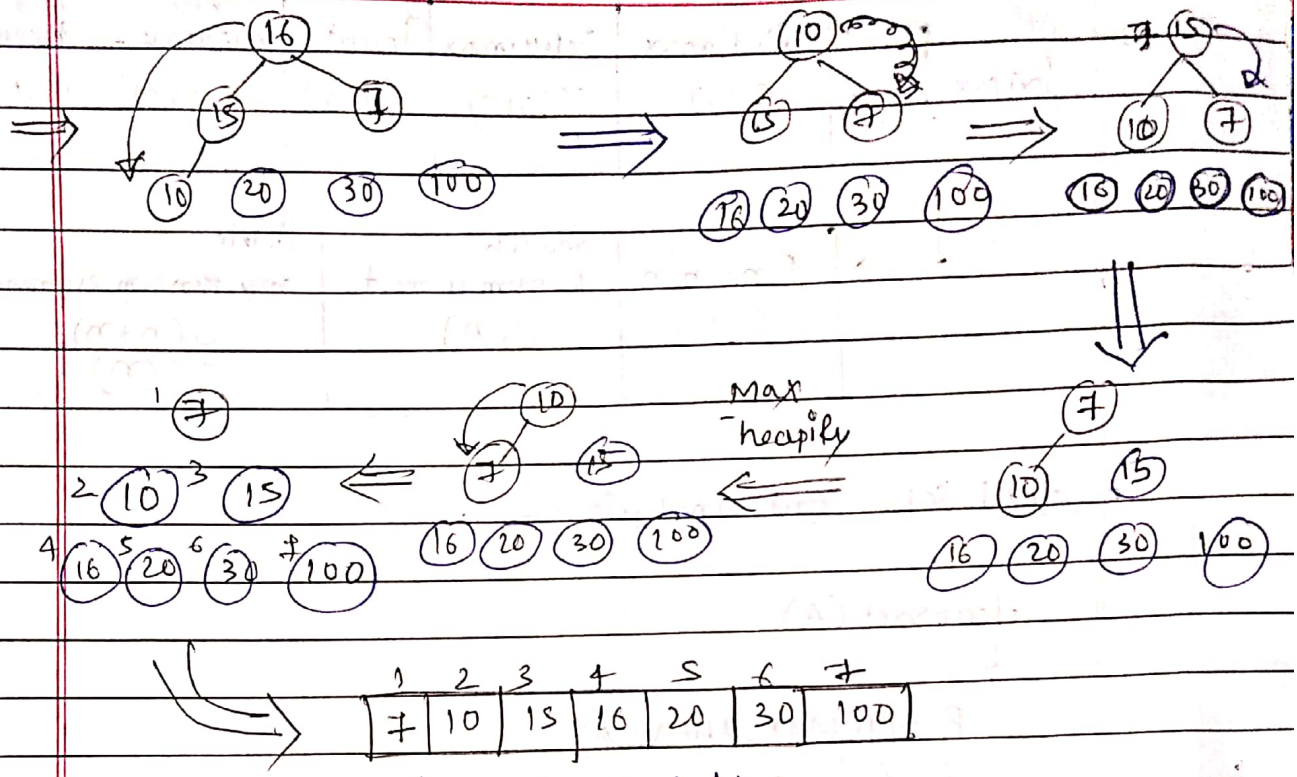
Find min	Search random element	delete any random element
O(n)	O(n)	O(n+n) = O(n)

- HEAP SORT and analysis =

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

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





→ Time complexity (Heap sort) = O(n log n)

T(log n) → for height of heap.
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(n log n) (b) θ(n) (c) θ(log n) (d) θ(1)

→ delete 1st min — O(log n) | Find 7th = O(1)
" 2 " | Insert = 6 * O(log n)
" 3 "
" 4 "
" 5 "
" 6th min — O(log n) | Time = 6 * O(log n) + O(1)
= O(log n)

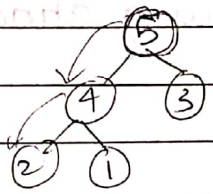
Question-②

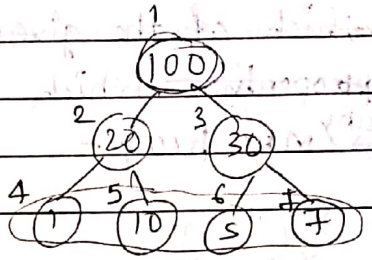
In a binary max heap containing 'n' numbers the smallest element can be found in time-

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

→

5	4	3	2	1





- smallest element

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

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

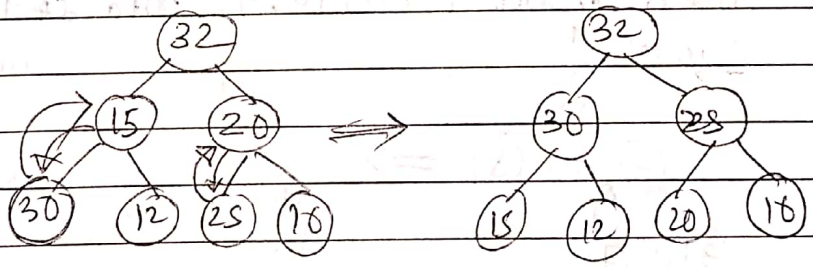
$= (4 \text{ to } 7)$

Time taken to find ~~min~~ 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,

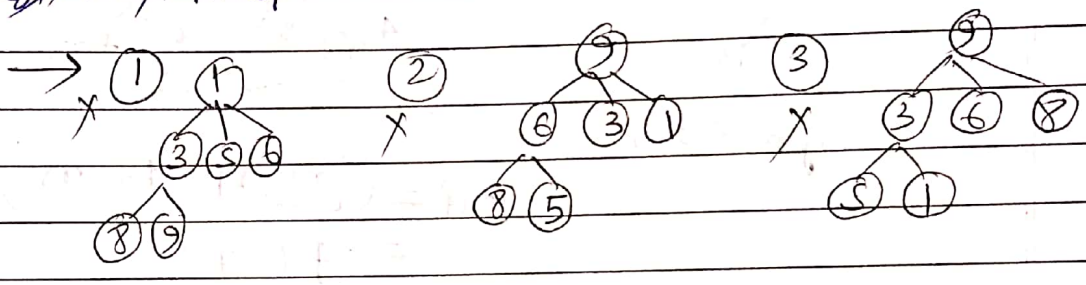
① 1, 3, 5, 6, 8, 9

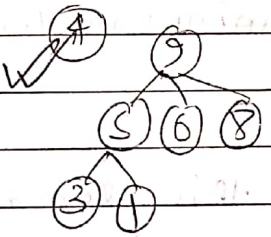
② 9, 6, 3, 1, 8, 5

③ 9, 3, 6, 8, 5, 1

④ 9, 5, 6, 8, 3, 1

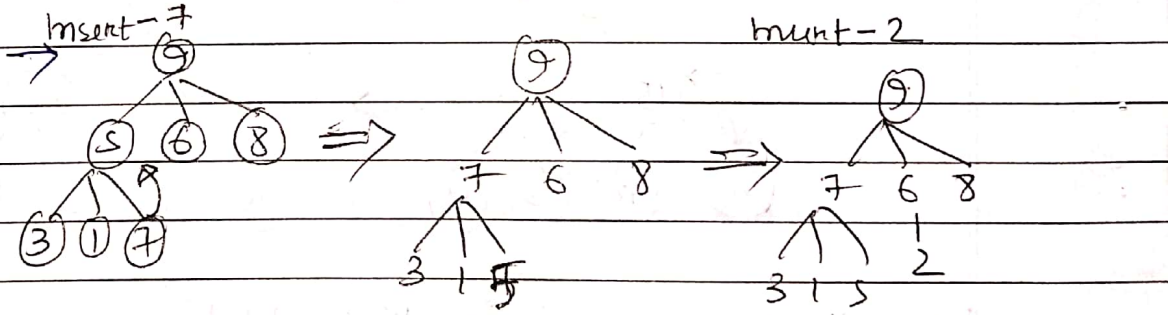
~~Which of the given sequence follow 3-ary Max heap property~~ — Which of the following array represent 3-ary Max heap —

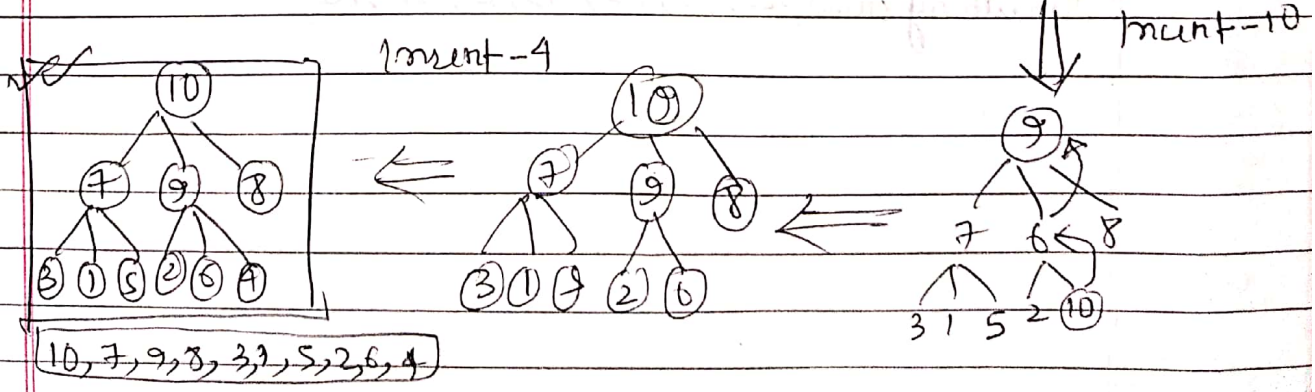




(always parent is greater than children)

then insert (7, 2, 10, 4) into ~~5~~ the result —



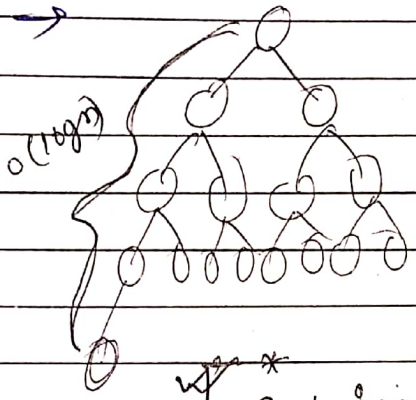


atlantis
Date
Page

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 O(log n)

= 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) = O(n)

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

for 'n' element BT.C = O(n)
2n " Time.C = O(2n) = O(n).

Greedy Algorithm

www.gatenotes.in

atlantis
Date
Page

- Introduction of Greedy algorithm —

Optimization :

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

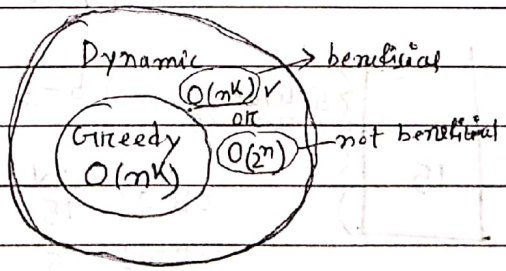
optimization Problems :
- minimize cost.
- max profit.
- maximize reliability.
- minimize risk.

→ Greedy method and dynamic programming are two programing peradene which could be used to solve optimization problem.

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

→ Dynamic program can solve any optimization problem. (we want to apply Dynamic programing if can do something better compare to exclusive exsustive search)

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



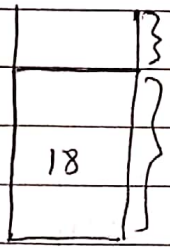
~~Greedy Knapsack algorithm~~

- Knapsack problem :
  - ↳ means (bag)

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

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

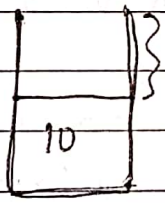
→ When Greedy about profit –



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

15W – 24P
2W = 24 * 2/15

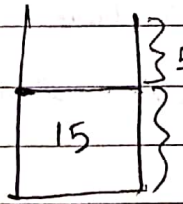
→ When Greedy about Weight –



	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 : 25/18 = 1.4
Ob-2 : 24/15 = 1.6
Ob-3 : 15/10 = 1.5



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

ratio of prof
When Greedy about more Profit compare to

atlantis
Date
Page

Algorithm =

```
Greedy Knapsack
{
    for i=1 to n;
    Compute Pi/wi;  — O(n)
    sort objects in non increasing order of P/w.
    for (i=1 to n from sorted list.
    if (m>0 && wi<=M)
        M = M - wi;
        P = P + Pi;
    else
        break;
    if (m>0)  — O(1)
        P = P + Pi(M/wi);
}
```

sorting algo take (merge sort) or heap sort O(n log n)

O(n)

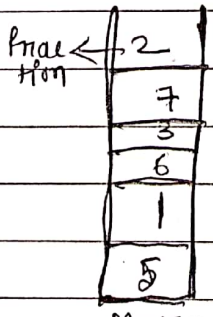
Ex: 1 What is max profit get out of it -

M = 15, n = 7

i =	1	2	3	4	5	6	7
Objects	1	2	3	4	5	6	7
profits	10	15	18	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



M = 15 units.

M = 15 13 12 8 3 (2) 0

P = 6 + 10 + 18 + 15 + 3 + 5 (2/3)

= (55.3)