Date: _____ Page No: ___

Books ⇒ ① Cormen
② Horowitz & Sahni

① Tanenbaum
② Kleinberg Tardos
} DS

Topics ⇒ ① Searching & Sorting
②

Page No.
17 -
-

Date: _____ Page No: ____

Logarithm:

$k^y = x \Rightarrow \log_k x = y$

eg $2^4 = 16 \Rightarrow \log_2 16 = 4$

* $\log_x a + \log_x b = \log_x ab$

* $\log_x a - \log_x b = \log_x \left(\frac{a}{b}\right)$

* $\log_x 1 = 0 \quad ; x > 1$

* $\log_a a = 1 \Rightarrow a^1 = a$

* $\log_x a^m = m * \log_x a$

eg $\log_2 2^3 = 3 \log_2 2 = 3$

* $\log_{x^k} a = \frac{1}{k} \log_x a$

* $\log_{x^k} a^m = \frac{m}{k} \log_x a$

* $\log_{a^k} a^m = \frac{m}{k} \log_a a = \frac{m}{k}$

* $\log_b a = \frac{1}{\log_a b}$ * $\log_b a . \log_a b = 1$

* $\log_a b = \frac{\log_x b}{\log_x a}$

eg $\log_2 12 = \frac{\log_{10} 12}{\log_{10} 2}$

* $x^{(\log_y k)} = k^{(\log_y x)}$

eg $2^{\frac{1}{2}\log_2 n} = 2^{\log_2 n^{1/2}} = (\sqrt{n})^{\log_2 2} = \sqrt{n}$

* $\log_a 0 = \begin{cases} -\infty & ; a > 1 \\ +\infty & ; 0 < a < 1 \end{cases}$

* $\log_a x^{2k} = 2k \log_a |x|$ $\quad \left\{ ; k \in I, x > 0, a \neq 1, k \neq 0, a > 0 \right.$

* $\log_{(a^{2k})} x = \frac{1}{2k} \log_{|a|} x$

* $\log_a x^2 = 2 \log_a |x|$

* $C_1 \cdot \log_2 n \leq 1 + \frac{1}{2} + \frac{1}{3} \cdots \frac{1}{n} \leq C_2 \cdot \log_2 n$

$C_1 \cdot \log_2 n \leq H_n \leq C_2 \log_2 n$

$\therefore H_n = \Theta(\log_2 n)$

→

1 2 3 4 5 ⇒ sequence

1 + 2 + 3 + 4 + 5 ⇒ series

Date: Page No:

Sequence & Series:

- 2 4 8 10 12 17 19 _ _ _ _ ⇒ not a sequence
- 1 3 9 27 _ _ _ ⇒ sequence
- 1/27 1/9 1/3 1 3 _ _ _ ⇒ sequence
- 1/2 1/3 1/4 1/5 _ _ _ ⇒ sequence

① AP : a $a+d$ $a+2d$ _ _ _ _ $a+(n-1)d$

- 1 4 7 10 13 _ _ _ _
  +3 +3 +3 +3

* n^{th} term of AP $(t_n) = a+(n-1)d$

* sum of first n terms $(S_n) = \frac{n}{2}[t_1 + t_n]$

eg 2 4 6 8 10 12 14.
① ② ③ ④ ⑤ ⑥ ⑦

* sum of k^{th} term from beginning & k^{th} term from last = 1^{st} term + last term

eg $2+14 = 4+12 = 6+10 = 8+8 = 16$.

$$S_n = \frac{n}{2}(a + a+(n-1)d) = \frac{n}{2}[2a+(n-1)d]$$

Que 3 no. are in AP. such that, $t_1 + t_2 + t_3 = 30$ & $t_1 \cdot t_2 \cdot t_3 = 960$. Find the nos.

⇒ if n is odd ⇒ $\alpha-\beta \quad \alpha \quad \alpha+\beta$ (c.d. = β)

⇒ if n is even ⇒ $\alpha-3\beta \quad \alpha-\beta \quad \alpha+\beta \quad \alpha+3\beta$ (c.d. = 2β)

⇒ n = 3 ⇒ $\alpha-\beta \quad \alpha \quad \alpha+\beta$

$\cdot\ \alpha-\beta+\alpha+\alpha+\beta = 30$

$3\alpha = 30$

$\alpha = 10$

$\cdot\ \alpha.(\alpha-\beta)(\alpha+\beta) = 960$

$\alpha(\alpha^2-\beta^2) = 960$

$10(100-\beta^2) = 960$

$\beta^2 = 4$

$\beta = \pm 2$

$\therefore$ 8 10 12

---

* If a, b, c are in AP, then,

$b = \frac{a+c}{2} \Rightarrow b-a = c-b$

$2b = a+c$

$b = \frac{a+c}{2}$, arithmetic mean

Date: Page No:

② GP : $a \quad ar \quad ar^2 \quad \_\_\_ \quad ar^{n-1}$

• 1 2 4 8 16 32 ...

* n^{th} term of GP $(t_n) = ar^{n-1}$

* $S_n = \begin{cases} \frac{a(r^n-1)}{r-1} & ; r > 1 \\ \frac{a(1-r^n)}{1-r} & ; r < 1 \\ na & ; r = 1 \end{cases}$

* $S_\infty = \begin{cases} \infty & ; r > 1 \\ \frac{a}{1-r} & ; r < 1 \\ \infty & ; r = 1 \end{cases}$

* If a, b, c are in GP, then

$$b = \sqrt{ac} \Rightarrow \frac{b}{a} = \frac{c}{b}$$

$$b^2 = ac$$

$b = \sqrt{ac}$, geometric mean

eg 3 6 12 24 48 96 192 384 - - -

eg 1 2 4 8 16 32.
① ② ③ ④ ⑤ ⑥

* product of k^{th} term from beginning and k^{th} term from last = product of first term and last term.

eg $1 * 32 = 2 * 16 = 4 * 8 = 32$

* product $(P_n) = (t_1 * t_n)^{n/2}$
of GP.

$= (a * ar^{n-1})^{n/2} = (a^2 r^{n-1})^{n/2}$

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

—x—

③ HP:

eg 1 1/2 1/3 1/4 ... 1/n

Date:_____ Page No:___

AGP

Que $S = 1 + 2x + 3x^2 + 4x^3$ _____ (∞ terms)

$\Rightarrow S = 1x^0 + 2x^1 + 3x^2 + 4x^3$ _ _ _

$xS = x + 2x^2 + 3x^3 + 4x^4$ _ _ _ | multiply with comm ratio.

$S - xS = 1 + x + x^2 + x^3$ _ _ _

$S(1-x) = \frac{1}{1-x}$; $x < 1$ (x)

$\therefore S = \frac{1}{(1-x)^2}$

⟶ —

Que $S = 1 + 2x + 3x^2 + 4x^3$ _ _ _ $t_{n-1} + t_n$ (n terms)

$xS = x + 2x^2 + 3x^3$ _ _ _ $+ x t_{n-1} + x t_n$

$S(1-x) = 1 + x + x^2 + x^3$ _ _ _ $x^{n-1} - x t_n$

$S(1-x) = \frac{(1-x^n)}{1-x} - x t_n$; $x < 1$

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

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

⟶

$S_n = \Sigma(\text{constant}) = n$

eg $\Sigma 2 = 2n$, $\Sigma 1 = n$ Date: Page No:

Que $S_n = 1 + 4 + 10 + 19 + 31 + 46 \_\_\_ t_n$

(AP) → 3, 6, 9, 12, 15

$\Rightarrow S_n = 1 + 4 + 10 + 19 + 31 + 46 \_\_\_ t_n$

$S_n = 1 + 4 + 10 + 19 + 31 + 46 \_\_ t_{n-1} + t_n$

$0 = 1 + 3 + 6 + 9 + 12 + 15 \_\_\_ (3n-3) - t_n$

$(n-1)$ terms

$\Rightarrow a = 3$, c.d. $= 3$ $\Rightarrow t_{n-1} = a + (n-2)d$

$= 3 + (n-2)3$

$= 3 + 3n - 6$

$= 3n - 3$

$\Rightarrow t_n = 1 + 3 + 6 + 9 \_\_\_ (3n-3)$ ... (i)

$t_n = 1 + 3(1 + 2 + 3 \_\_\_ (n-1))$

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

$= 1 + \frac{3(n^2-n)}{2}$

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

$S_n = \Sigma \frac{1}{2}(2 + 3n^2 - 3n) = \frac{1}{2}(\Sigma 2 + 3\Sigma n^2 - 3\Sigma n)$

$= \frac{1}{2}\left(2n + \frac{3n(n+1)(2n+1)}{6} - \frac{3n(n+1)}{2}\right) = \frac{n^3+n}{2}$

Product ($\prod$) $\Rightarrow \prod_{i=1}^{n} k^i = k^n * n!$

Date: Page No:

* $\sum_{r=1}^{n} r = 1 + 2 + 3 \ldots n = \frac{n(n+1)}{2}$

* $\sum_{r=1}^{n} 1 = 1 + 1 + 1 \ldots n = n$

* $\sum_{r=1}^{n} r^2 = 1^2 + 2^2 + 3^2 \ldots n^2 = \frac{n(n+1)(2n+1)}{6}$

* $\sum_{r=1}^{n} r^3 = 1^3 + 2^3 + 3^3 \ldots n^3 = \left[\frac{n(n+1)}{2}\right]^2$

→

* $t_n = 2n - 3 \Rightarrow$ linear term $\Rightarrow$ A.P.

$\therefore t_1 = (2 * 1) - 3 = -1$

$\therefore c.d = t_n - t_{n-1} = 2n - 3 - 2(n-1) + 3 =$

$= 2n - 3 - 2n + 2 + 3$

$= 2$

* $S_n = \Sigma t_n$

$S_n = \Sigma (2n - 3)$

$= \Sigma 2n - \Sigma 3$

$= 2\Sigma n - 3\Sigma 1$

$= 2 * \left[\frac{n(n+1)}{2}\right] - 3n = n^2 - 2n$

←

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

Date: Page No:

Que $S_n = 2 + 5 + 10 + 19 + 36 + 69 \ldots$

3 5 9 17 33 ..

GP → 2 4 8 16 ..

$\Rightarrow S_n = 2 + 5 + 10 + 19 + 36 + 69 \ldots t_n$

$S_n = 2 + 5 + 10 + 19 + 36 \ldots t_{n-1} + t_n$

$0 = 2 + 3 + 5 + 9 + 17 + 33 \ldots T_n - t_n$

$t_n = 2 + 3 + 5 + 9 + 17 + 33 \ldots T_n$

$t_n = 2 + 3 + 5 + 9 + 17 \ldots T_{n-1} + T_n$

$0 = 2 + 1 + 2 + 4 + 8 + 16 \ldots (\quad) - T_n$

($n-1$) terms

$T_n = 2 + \left(\frac{2^n - 1}{2-1}\right) = 2 + 2^n - 1 = 2^n + 1$

$\Rightarrow t_n = \Sigma T_n$

$= \Sigma(2^n + 1)$

$= \Sigma 2^n + \Sigma 1$

$= 2^{n+1} - 1 + n$

$\Rightarrow S_n = \Sigma t_n$

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

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

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

$\rightarrow S_n = \frac{2^{n+3} - 2 - 8n + n^2}{2}$

Date: Page No:

Recursion:

```
* void P()                    void main()
  { int a=1;                  { P();
    printf("%d", a);            P();          O/P ⇒ 1 1 1
    a=a+1;                      P();
  }                           }
```

* static int a = 1; ⇒ 1 2 3

* static int a; ⇒ 1 1 1
  a=1;

```
* void P(int n)
① { if (n<=0)                 void main()
②     return;                 {
③   else                        P(4);        ⇒ 1 2 3 4
④   { P(n-1);                  }
⑤     printf("%d", n);
    }
}
```

stack :

| n=4 | n=3 | n=2 | n=1 | n=0 | ← auto var.
| ⑤ | ⑤ | ⑤ | ⑤ | | ← address
| P(4) | P(3) | P(2) | P(1) | P(0) |

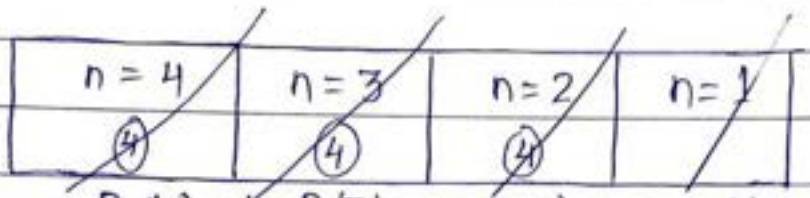
---

* Recursive programs have higher time complexity than iterative programs (loops)

Date:_____ Page No:_____

*
```
① {  int P(int n)
②    if (n<=1)
③    return 1;
④    else
     return n + P(n-1);
   }
```

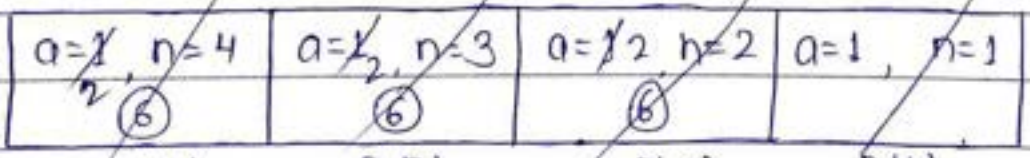
```
void main()
{  int y = P(4);
   printf("%d", y);
}
```

⇒ 

⇒ 10

*
```
① {  int P(int n)
     int a = 1;
②    if (n<=1)
③    return 1;
④    else
⑤    { a = a+1;
⑥    return a + P(n-1);
     }
   }
```

```
void main()
{
   int y = P(4);
   printf("%d", y);
}
```

⇒ 

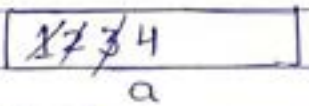
⇒ 7

* Recursive codes have higher space complexity than that of iterative because of activation records of func calls in stack.

Date: Page No:

in above code,

* static int a = 1;



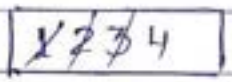
⇒

n=4	n=3	n=2	n=1
⑥	⑥	⑥	
P(4)	P(3)	P(2)	P(1)
4+9=13	4+5=9	4+1=5	1

⇒ 13

```
int P(int n)
{
① static int a = 1;
② if (n<=1)
③  return 1;
④ else
⑤ { a=a+1;
⑥  return a+n+P(n-1);
  }
}
```

```
void main()
{
int y=P(4);
printf("%d", y);
}
```



⇒

n=4	n=3	n=2	n=1
⑥	3+3+P(2)	4+2+P(1)	
P(4)	P(3)	P(2)	P(1)
2+4+13=19	3+3+7=13	4+2+1=7	1

→ return 2+4+P(3)

⇒ 19

* '+' is left associative

a+n+P(n-1)

available

(a+n) + P(n-1)

---

in above code,

* return a+P(n-1)+n ⇒ 22

Date: _____ Page No: ____

* How many recursive calls to compute P(4)?

⇒ P(4)
P(3)
P(2)
P(1)
} ⇒ 3 recursive calls.

---

* void P(int n)
{ if (n<=0) return;
else
{ P(n-1);
printf("%d", n);
}
}

void main()
{ P(4);
}

P(4)
P(3)
P(2)
P(1)
P(0)
} ⇒ 4 rec. calls

---

* void P(int n)
{ P(n-1);
printf("%d", n);
}

P(4);
⇒ ∞ rec. calls.

---

* ① while(1)
{
printf("%d", n);
}

② * void P(int n)
{
P(n-1);
printf("%d", n);
}

* Time complexity of recursive code ⇒ No. of rec. calls * complexity of 1 rec. call

* If recurrence rel. is tough to solve ← make tree.

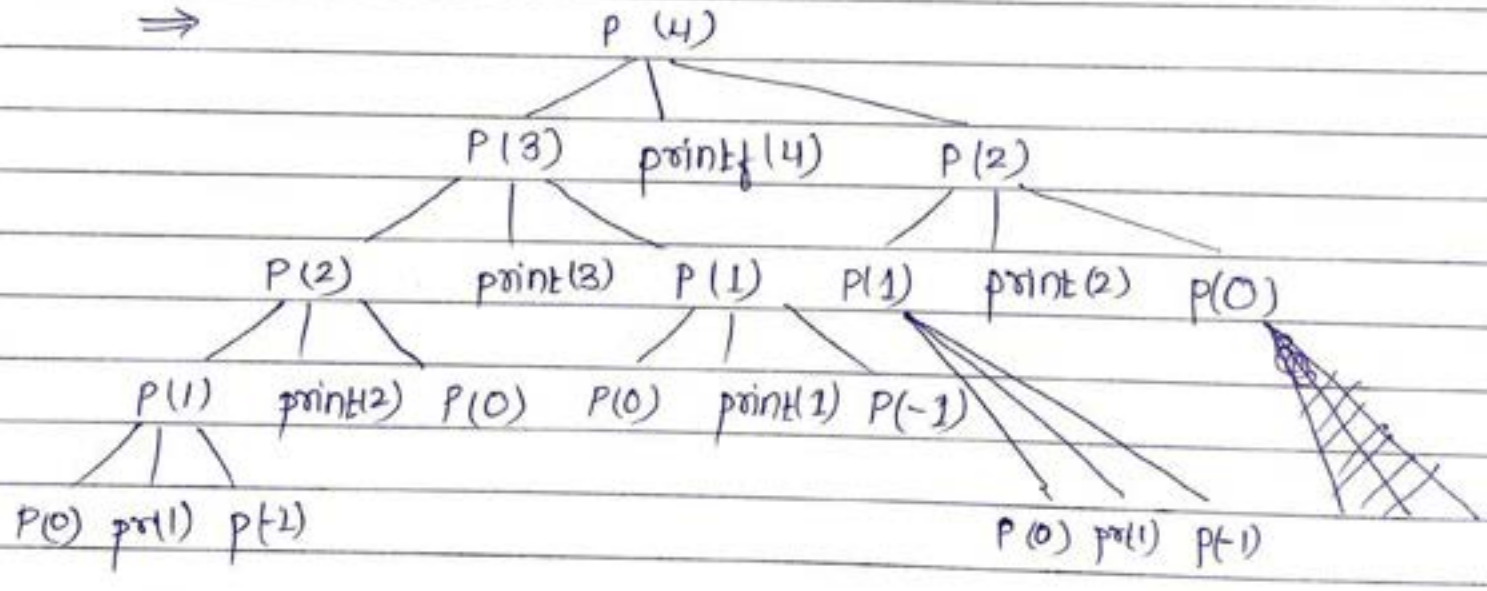
Date Page No.

* ① program is an infinite code but ② is not, as at some point of time, stack will be full.

```
* void P(int n)
  {  if(n<=0)
       return;
     else
     {  P(n-1);
        printf("%d", n);
        P(n-2);
     }
  }
```

```
void main()
{
   P(4);
}
```

⇒



⇒ 14 recursive calls.

O/P ⇒ 1 2 3 1 4 1 2

Depth of Recursion ⇒ n+1 (5 for P(4))

∴ Space complexity ⇒ O(n)

Date: _____ Page No: _____

* Perform DFS of recursion tree from L to R for the O/P of code:

CS-2004 Que.

```
int f (int n)
{
①   static int i=1;
②   if (n≥5) return n;
③   n=n+i;
④   i++;
⑤   return f(n);
}
```

i = 1 2 3 4

n=2	n=4	n=7	n=7
⑤	⑤	⑤	
f(1)	f(2)	f(3)	

Ans ⇒ 7

CS-2007 Que 42

```
int f (int n) {
①   static int r=0;
②   if (n<=0) return 1;
③   if (n%3)
④   { r=n;
⑤   return f(n-2)+2;
⑥   }
⑦   return f(n-1)+r;
}
```

r = 0 5

n=5	n=3	n=2	n=1	n=0
f(3)+2	f(2)+r	f(1)+r	f(0)+r	
f(5)	f(3)	f(2)	f(1)	f(0)
16+2=18	11+5=16	6+5=11	1+5=6	

Ans → 18

* Space complexity ⇒ Depth of longest branch of recursion tree.

SEARCHING & SORTING

Date: Page No.

① Bubble Sort :

eg 10, 6, 12, 18, 3, 1 ⇒ 5 comparisons

⇒ Pass ⇒ ① ② ③ ④ ⑤

①	②	③	④	⑤
10	6	6	3	1
6	10	10	3	1
12	12	3	1	3
18	3	1	6	6
3	1	12	10	10
1	18	18	12	12
			18	18

* After ① pass, largest element is at its position.
* In ① pass, there are (n-1) comparisons if no. of elements is n.
* In ② pass, there are (n-2) comparisons. & so on.
* There are (n-1) passes in bubble sort.
* no. of comparisons $= (n-1)+(n-2) \dots 3+2+1$

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

$$= \frac{n^2-n}{2}$$

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

eg

	①	②	③	④	⑤	⑥
40	40	12	12	5	5	5
60	12	40	5	10	9	9
12	60	5	10	9	10	10
80	5	10	9	12	12	12
5	10	9	40	40	40	40
10	9	60	60	60	60	60
9	80	80	80	80	80	80

* no of swaps ⇒ 15

-x-

* max. no. of swaps in bubble sort = $\frac{n(n-1)}{2}$

eg 6 5 4 3 2 1 ⇒ swaps = 15

→

* min. no. of swaps in bubble sort = 0

eg 1 2 3 4 5 6 ⇒ swaps = 0

→

Code :

```
void bubbleSort (int a[], int n)
{ int i, j, c ;
  for(i=0; i<n-1; i++)        // no. of passes
  {
    for(j=0; j<(n-1-i); j++)  // no. of comparisons
    {
      if (a[j] > a[j+1])
      { c = a[j]; a[j] = a[j+1]; a[j+1] = c; }
    }
  }
```

Date: Page No:

* If, in any pass, there's no swap, we can say that array has been sorted. by modifying it.

eg 6 10 12 18 20 22 (already sorted)

1st pass i.e. if there's no swap,

* After (n-1) comparisons we'll say array is sorted.

Best case ⇒ n-1 comparisons ⇒ O(n)

Worst case ⇒ $\frac{n(n-1)}{2}$ comparisons ⇒ $O(n^2)$

—x—

⇒ Thus, we can modify bubble sort to have O(n) B in best case.

② Selection Sort : let, min = 20

eg

	①	②	③	④	⑤
20	1	1	1	1	1
9	9	3	3	3	3
22	22	22	8	8	8
1	20	20	20	9	9
3	3	9	9	20	20
8	8	8	22	22	22

* After ① pass, smallest element is at its position.

* In ① pass, there are (n-1) comparisons

* In ② pass, there are (n-2) comparisons & so on,

* There are (n-1) passes in selection sort.

* In each pass, there can be only one swap or zero.

* no. of comparisons = $\frac{n(n-1)}{2}$ $\Rightarrow$ $O(n^2)$

* In bubble sort, if there's no swap in first pass, the array is already sorted but it is not the case with selection sort.

Date:_____ Page No:____

* min. no. of swaps = 0

* max. no. of swaps = (n-1) [per pass, 1 swap]

Code:

```
void selectionsort (int a[], int n)
{
  for( i=0; i<(n-1); i++)        // no. of passes
  {  min = a[i]; k = i;
     for( j= i+1; j<= n-1; j++)  // no. of comparisons
     {
       if (a[j] < min)
        {  min = a[j]; k = j; }
     }
       c = a[i];
       a[i]= a[k];
       a[k] = c;
  }
}
```

Date: _____ Page No: _____

Types of Sorting Algorithms :

① • Comparison Based Sorting ⇒ Bubble, Selection, Insertion, Heap, Quick, Merg

• Counting Based Sorting ⇒ Radix, Counting.

② • Stable Sorting ⇒ An algo. is stable if it maintains relative order of repeated data.

eg 9^1 10 9^2 5 6

Bubble → ① 9^1 9^2 5 6 10
② 9^1 5 6 9^2 10
③ 5 6 9^1 9^2 10

* Bubble sort is stable.

Selection ⇒ ① 9^1 9^2 3 1
① 1 9^2 3 9^1

* selection sort is unstable.

③ • Inplace Sorting : An algo. is in-place if it doesn't require an additional data str./array to sort the elements.

* All comparison based sortings except merge sort are inplace.
* All counting based sorting are not in place.

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2)$
$< O(n^3) < O(n^k) < O(2^n)$ Date:____ Page No:____
(polynomial) (exponential)

④ · Internal Sorting: It can sort data available in main memory only.

· External Sorting: It can sort data available in secondary memory.

* Only merge sort can be used as external also.

—x—

Time Complexity

	Best	Average	Worst
① Bubble Sort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
② Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
③ Insertion Sort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
④ Heap Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
⑤ Quick Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n^2)$
⑥ Merge Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
⑦ Radix Sort	$\Theta(nk)$	$\Theta(nk)$	$\Theta(nk)$
⑧ Counting Sort	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

	Space Comp.	Stable
① Bubble Sort	$O(1)$	✓
② Selection Sort	$O(1)$	×
③ Insertion Sort	$O(1)$	✓
④ Heap Sort	$O(1)$	×
⑤ Quick Sort	$O(n)$	×
⑥ Merge Sort	$O(n)$	✓
⑦ Radix Sort	$O(n+k)$	✓
⑧ Counting Sort		×

* For smaller values of n, bubble, selection, insertion should be preferred. as they can be implemented easily.

* We select the algorithm which gives best complexity in worst case. (i.e merge & heap here)

* Heap sort is preferred over merge sort because merge is not in-place.

* But, Quick Sort is preferred over these two because,

① Worst case of Quick sort is only when array is already sorted; which is very rare.

* Time complexity shows the growth in time requirement, not actual time requirement.

eg Algo ① ⇒ O(n) Algo ② ⇒ O(n)

$n=1 \Rightarrow$ 2 hours $n=1 \Rightarrow$ 1 sec.

$n=10 \Rightarrow$ 20 hrs $n=10 \Rightarrow$ 10 sec

* Both are O(n), but Algo ② is faster.

② We know that time complexity of heap sort is less than that of quick sort; but, the time requirement of heap sort is greater than that of quick sort

∴ Quick sort > Heap sort > Merge Sort

Date: _____ Page No: _____

(3) Insertion Sort:

* In insertion sort, we assume that we've a sorted array & we insert the elements in that array.

eg

	⇒		①	②	③	④	⑤	⑥
20	⇒	20	6	6	6	5	5	5
6			20	12	10	6	6	6
12				20	12	10	7	7
10					20	12	10	10
5						20	12	12
7							20	19
19								20

* Initially, sorted array contains only 1 element & then an element is inserted per pass.

* Search from bottom until you get smaller element.

* There are (n-1) passes in insertion sort.

eg
1
2
3
4
5
6

* there will be one comparison in each pass in best case.

∴ (n-1) comparisons ⇒ O(n)

* No shifting is required in best case

eg
6
5
4
3
2
1

* 1 comparison in (1) pass.
* 2 comparisons in (2) pass.
* n comparisons in n^{th} pass

$\Rightarrow 1 + 2 + 3 \ldots (n-1)$ (passes) $= \frac{n(n-1)}{2}$

$\Rightarrow O(n^2)$

Que 10 12 16 5 3 20 21

* This is status of an array after some pass of a sorting algo. then that algo. might be;

(1) Bubble — last element is at its position.

(2) Selection — first element is not at its position.

(3) Insertion — first two elements are sorted.

Que How many comparisons are performed by insertion sort?

$\Rightarrow$ 19 5 12 6 18 7 9

(1) 19
(2) 5 19
(3) 5 12 19
(4) 5 6 12 19
(5) 5 6 12 18 19
(6) 5 6 7 12 18 19

Ans $\Rightarrow$ 16 comparisons

Date: Page No:

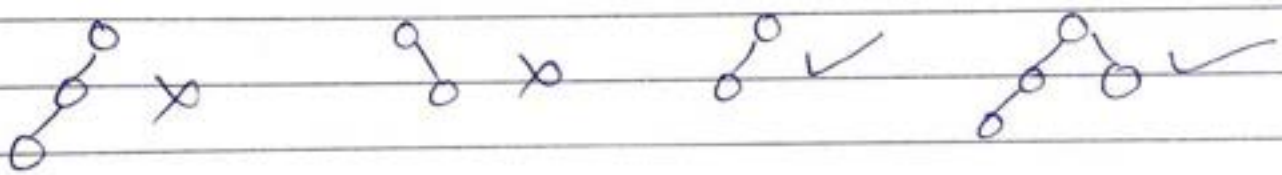
* Insertion sort is preferable if only few elements are out of order.

④ Heap sort :

* It is based on heap data structure.

* Heap is almost complete binary tree in which levels are filled from left to right.

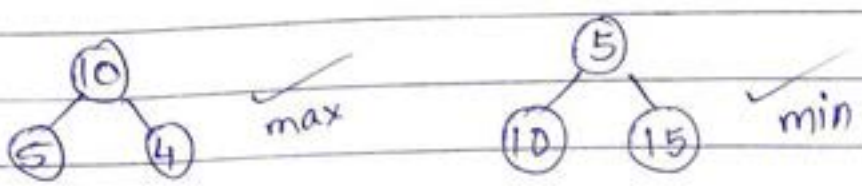
* You can switch to the next level only when previous level is completed.

eg 

* There exists 2 types of heaps;

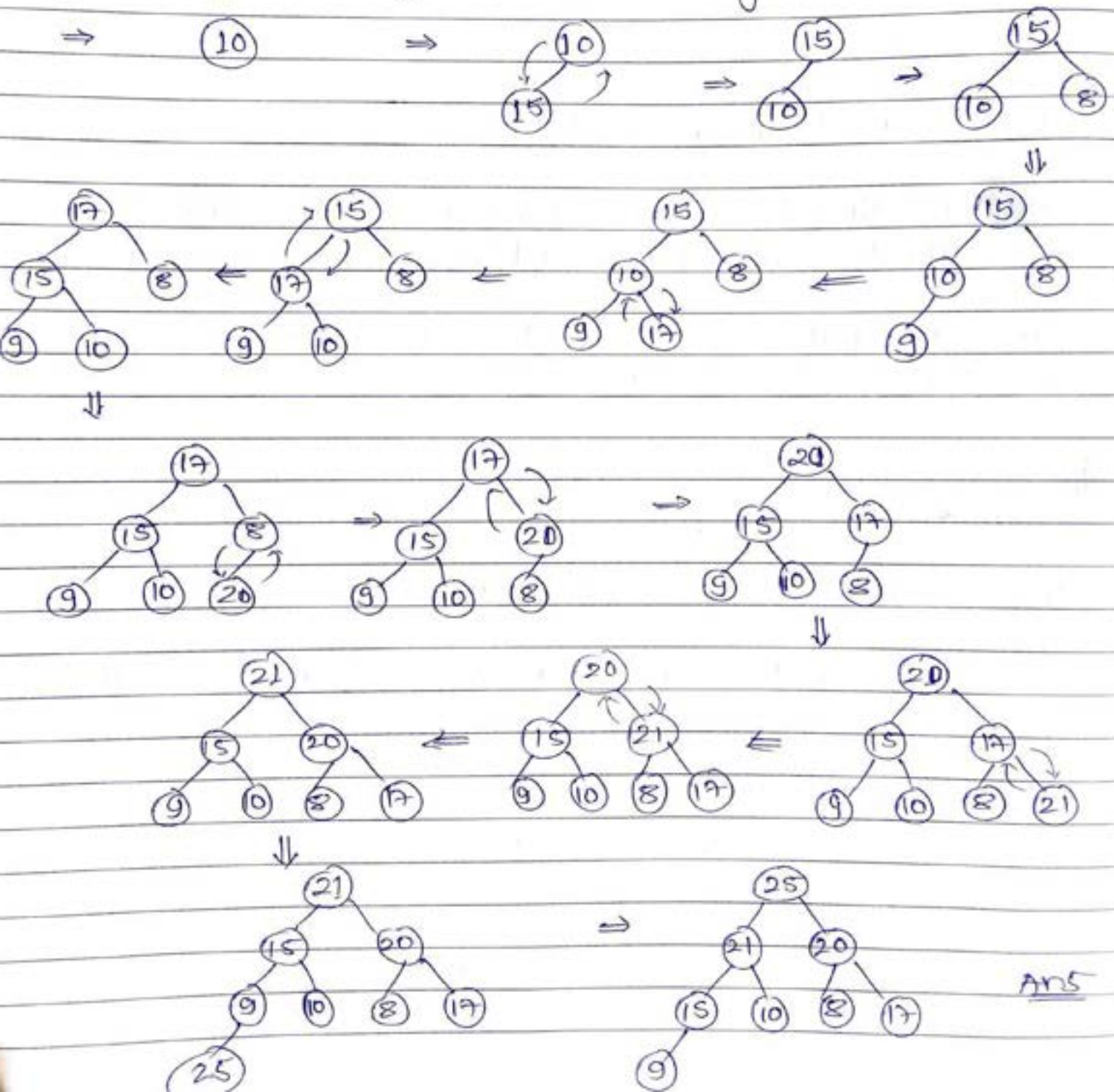
① Min Heap : Value of parent is less than or equal to that of child.

② Max Heap : Value of parent is greater or equal to that of child.

eg 

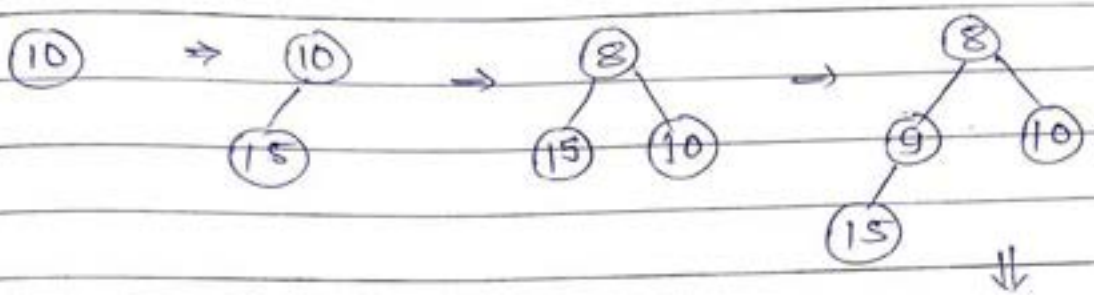
eg 10 15 8 9 17 20 21 25 ; create max heap.

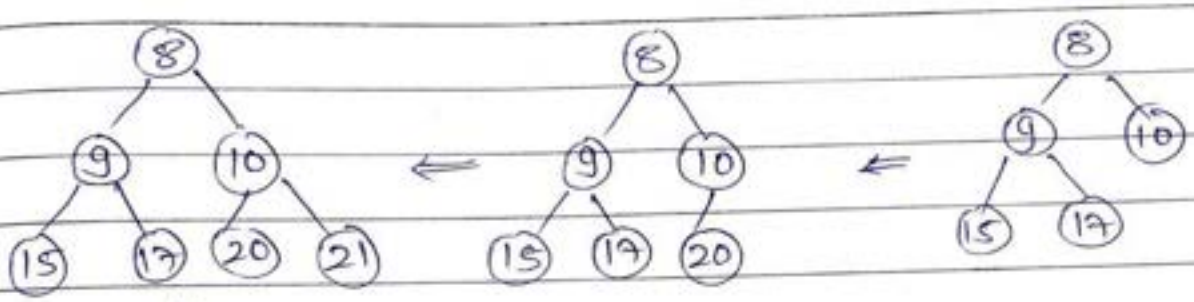
Algo (1) By inserting elements one by one

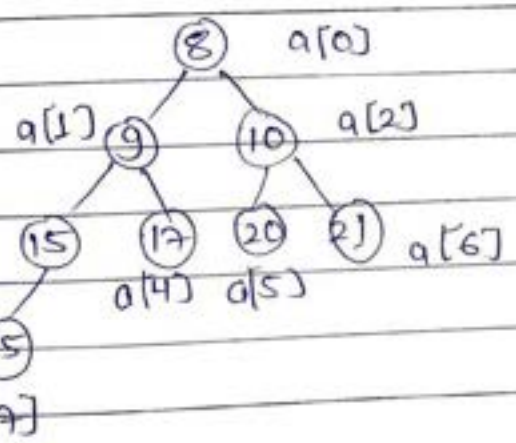


Date: Page No:

eg 10 15 8 9 17 20 21 25 ; create min heap.







No. of comparisons ⇒ 15-10, 10-8, 9-15
9-8, 17-9, 10-20
10-21, 15-25

⇒ 8 comparisons

⇒ How to store a binary tree in an array :

8	9	10	15	17	20	21	25
0	1	2	3	4	5	6	7

a:

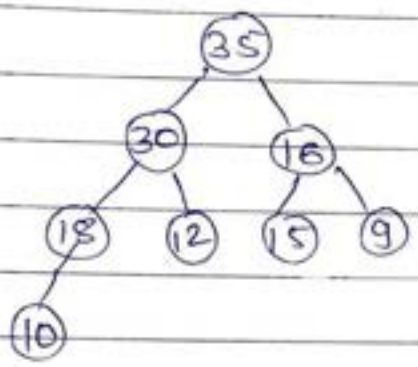
* level wise

* when index starts from 1 (not from 0) $\Rightarrow \left\lfloor \frac{c}{2} \right\rfloor, 2p, 2p+1$

Date: Page No:

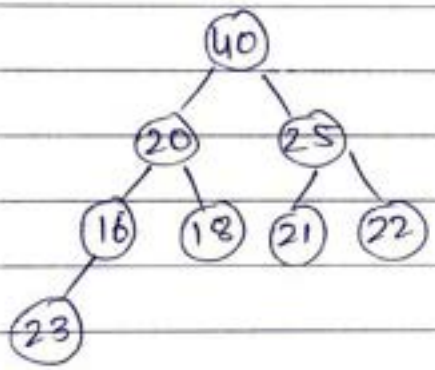
Que Is it a max heap?

(a) $\Rightarrow$ 35 30 16 18 12 15 9 10



Ans : Yes

(b) 40 20 25 16 18 21 22 23



Ans : No

$\rightarrow$ —

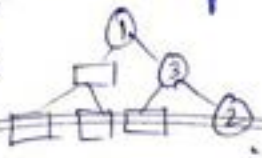
eg

a	40	20	25	16	18	21	22	23
0	0	1	2	3	4	5	6	7

* If p is the index of parent then, the index of left child will be (2p+1). & the index of right child will be (2p+2).

* If c is the index of child then, the index of parent will be $\left\lfloor \frac{c-1}{2} \right\rfloor$

* Min. size of array to store any binary tree of n nodes:

⇒ 

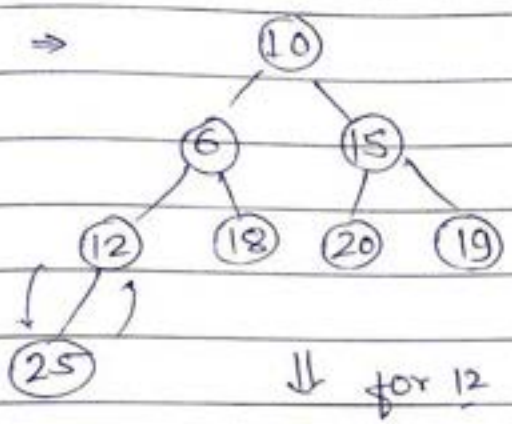
n = 3 ⇒ 7 ⇒ $2^n - 1$

1	3			1		2

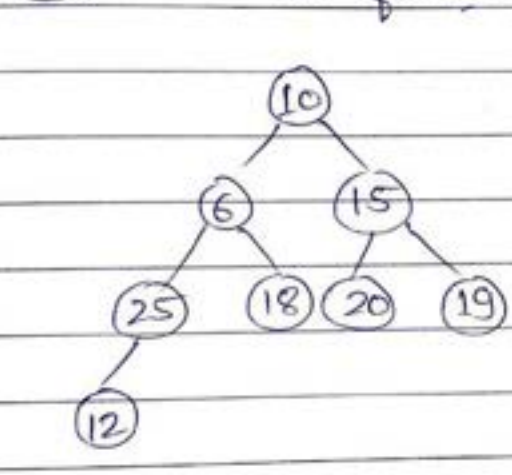
Date: Page No:

Algo (2): By inserting all elements at once (direct conversion of array into heap)

⇒ 10 6 15 12 18 20 19 25 ; max heap.

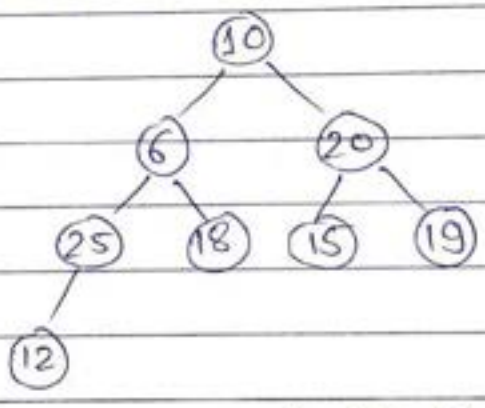
⇒ 

* Make a binary tree
* Start with first non-leaf node from bottom. & compare with its child nodes.

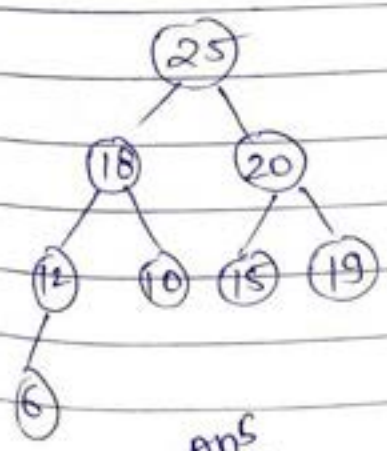


⇓ for 12

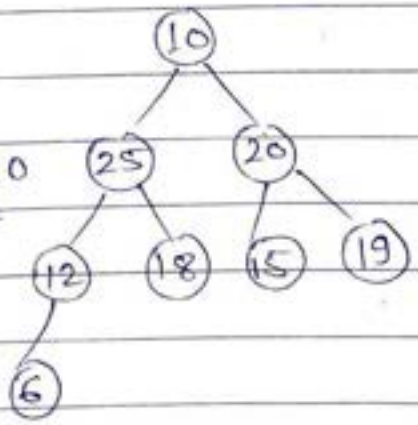
for 15 ⇒



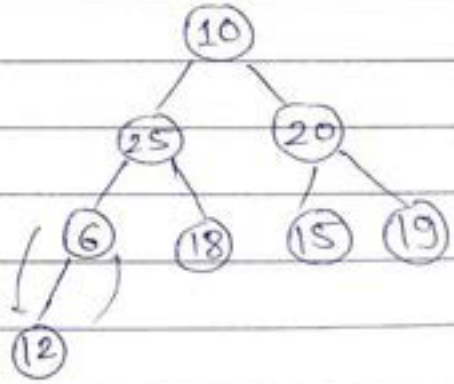
⇓ for 6



for 10 ⇐



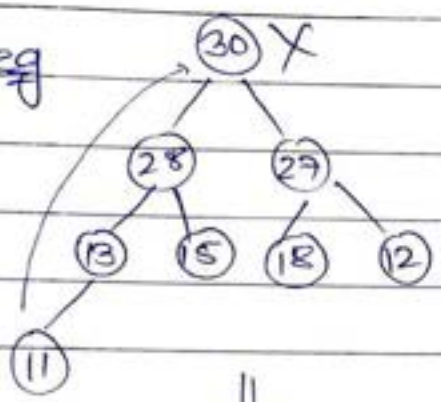
⇐



Ans

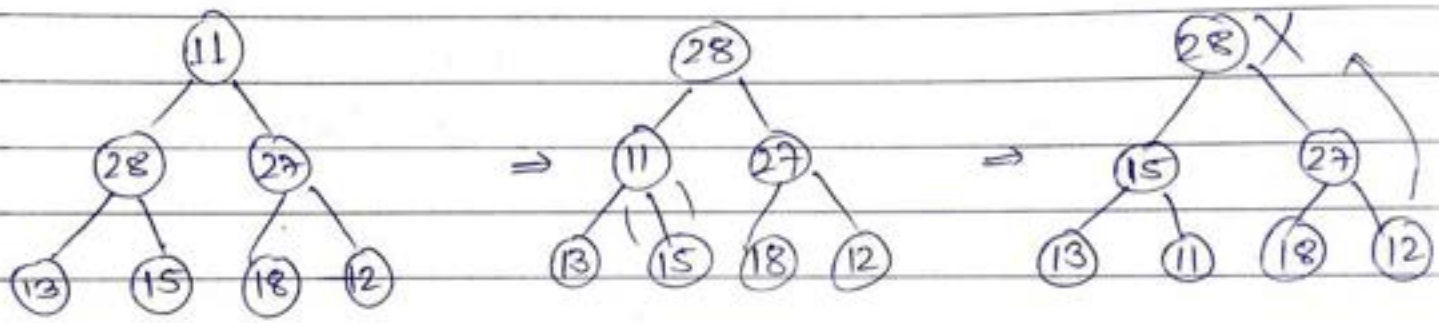
* Both algo. return different heaps.

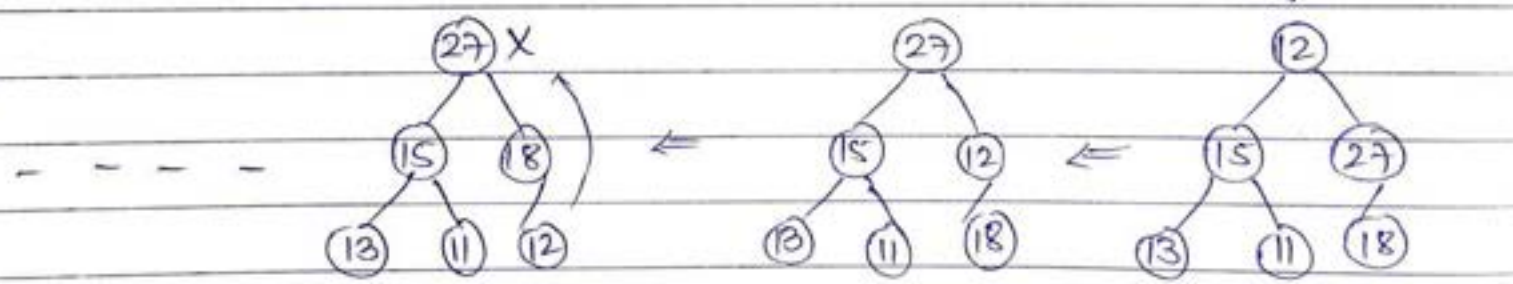
⇒ How to perform deletion from a heap!

eg 

* Deletion will always be started from root node.

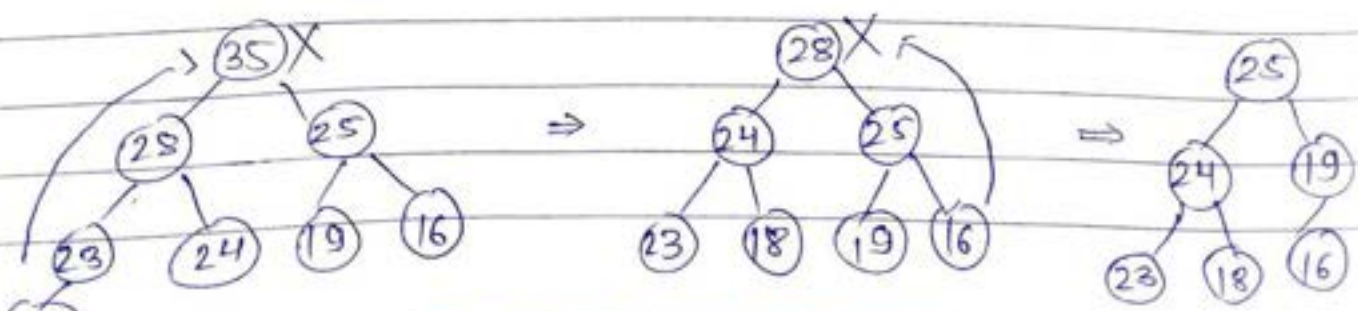
* Replace the root by last node ~~element~~ of heap & then heapify (if needed.).





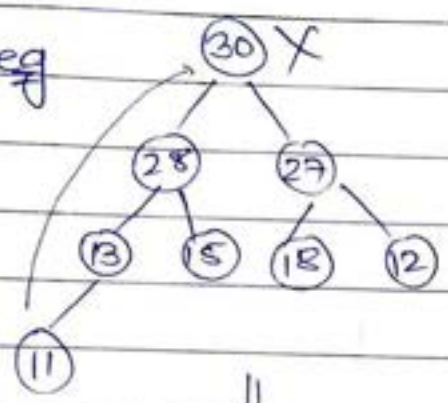
* Qu: what will be the status of heap after 2 deletions

⇒ 35 28 25 23 24 19 16 18



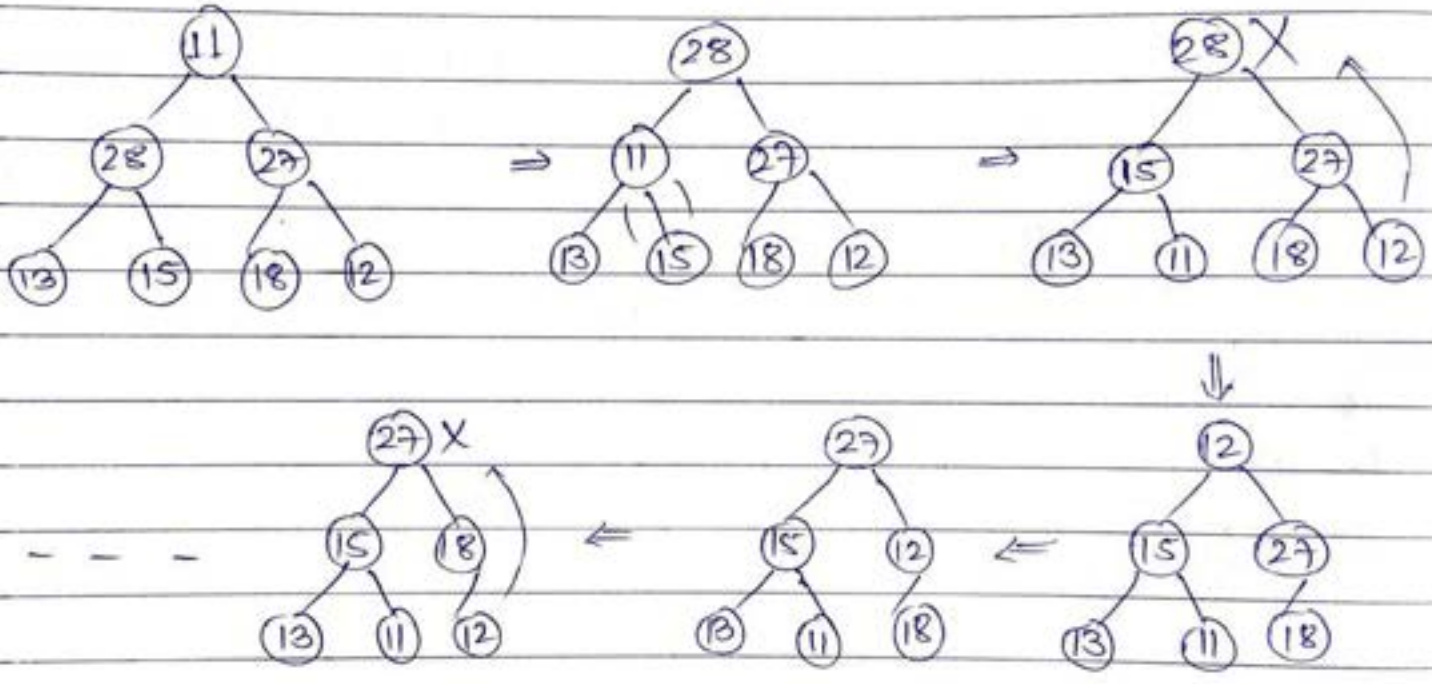
⇒ How to perform deletion from a heap!

eg



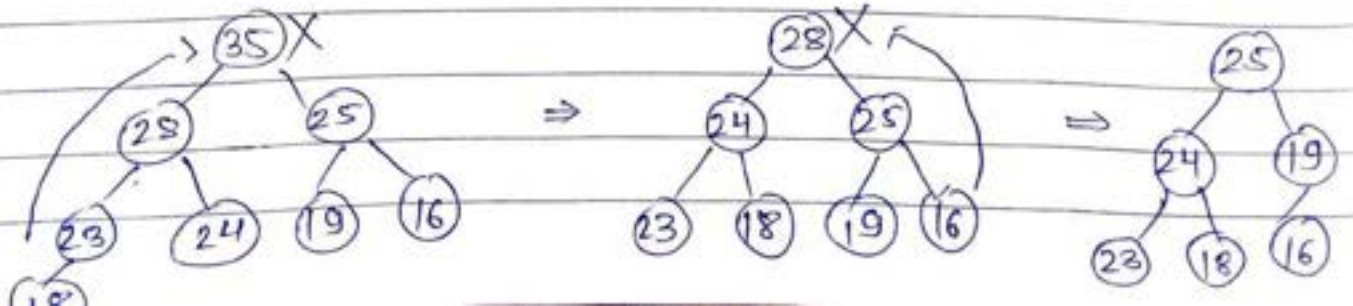
* Deletion will always be started from root node.

* Replace the root by last node ~~element~~ of heap & then heapify (if needed.).



Qu: what will be the status of heap after 2 deletions

⇒ 35 28 25 23 24 19 16 18



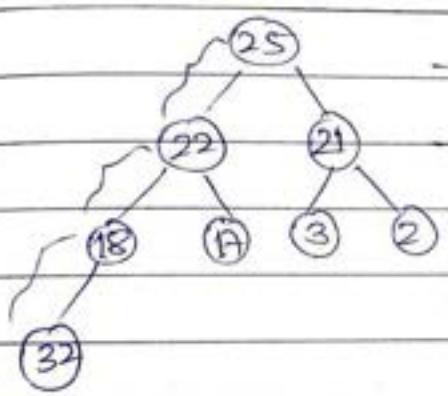
* we perform heap sort by; ① creating a heap of given array, ② Delete root node elements one by one

Ans ⇒ 25, 24 19 23 18 16

* Complexity to insert an element in heap :

⇒ max. No. of comparisons to insert an element in heap ⇒ $\lceil$(depth / height of heap) + 1$\rceil$

eg Insert 32, in given heap.



--- 2^0 node at 0^{th} level

--- 2^1 nodes at 1^{st} level

--- 2^2 nodes at 2^{nd} level

$\therefore 2^d$ nodes at d^{th} level.

* If there are n elements in heap, then depth of heap will be,

$$\Rightarrow 2^0 + 2^1 + 2^2 \text{ - - - } 2^d = n$$

$$\Rightarrow 2^0 \cdot \left(\frac{2^{d+1} - 1}{2-1}\right) = n \qquad \left[\frac{a(r^n-1)}{r-1}\right]$$

$$2^{d+1} = n+1$$

$$\log_2 2^{d+1} = \log_2 (n+1) \quad \text{(taking log)}$$

* Operations which are independent of size of data str. & take constant time, have O(1) time complexity.

$$d+1 \ (\log_2 2) = \log_2 (n+1)$$

$$d = \log_2 (n+1) - 1$$

$$d \cong \log_2 n$$

* If a heap has n nodes, depth of that heap will be logn.

∴ no. of comparisons to insert an element $= d = \log_2 n$

∴ complexity of inserting an element in heap $= O(\log n)$

→

* Complexity of finding the largest element in max heap

⇒ O(1) (no comparisons) (root)

* Complexity of deleting the largest element from max heap:

⇒ O(1) but reheapification will take O(logn)

* for insertions in heap, comparisons are performed from bottom to top & for deletions, top to bottom.

* for reheapification, each level has 2 comparisons
∴ no. of comparisons ⇒ 2logn ⇒ O(logn)

Date: _____ Page No: ____

* Complexity to delete second largest element : from max heap:

⇒ $O(\log n) + O(\log n) = O(2\log n) = O(\log n)$

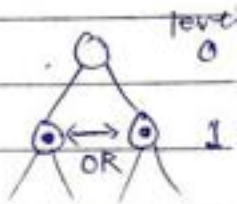
* Complexity to delete k^{th} largest element from max heap:

⇒ $O(k \log n)$; if k = constant ⇒ $O(\log n)$

—x—

* Complexity to find second largest element in max heap:

⇒ O(1) (1 comparison) (level 1)

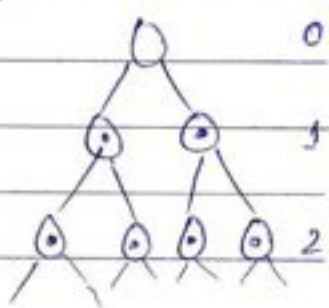


* Complexity to find third largest element in max heap:

⇒ O(1) (6 comparisons) (level 1 or 2)

* Complexity to find k^{th} largest element in max heap:

⇒ O (1)



—x—

* k^{th} largest element can be in level 1 to (k-1).

* Complexity of finding smallest element in max heap:

* smallest element will always be a leaf node.

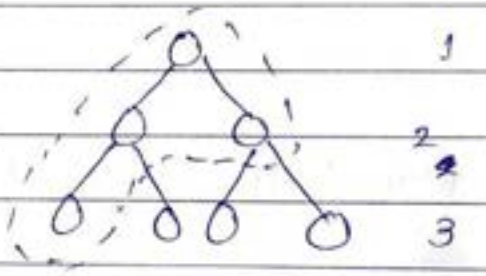
* Approx. half of the nodes in a heap are leaf nodes.

∴ no. of comparisons = $\frac{n}{2}$ ⇒ $\underline{\underline{O(n)}}$

* Leaf nodes are always on last or 2nd last level.

* Min. no. of elements in a heap of height n.
(consider root at height 1)

⇒ 2^{n-1}



n = 3 ⇒ min = 4
max = 7

* Max. no. of elements in a heap of height n.

⇒ $2^n - 1$

Date _____ Page No _____

* Complexity of Heap Creation:

Algo ① (one by one) ⇒ n elements * complexity of Inserting one element (i.e logn)

⇒ O(nlogn)

Algo ② ⇒ O(n)

* Complexity of Heap Deletion:

⇒ n elements * Complexity of deleting one element (i.e logn)

⇒ O(nlogn)

∴ Complexity of Heap Sort = Heap Creation + Heap Deletion

Algo ① ⇒ O(nlogn + nlogn)

⇒ O(nlogn)

Algo ② ⇒ O(n + nlogn) ⇒ O(nlogn)

Algorithm of Heap Sort :

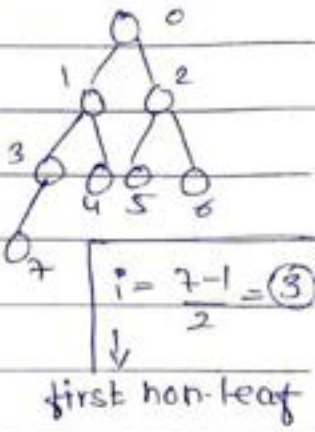
```
void max_heapify (int a[], int i, int size)
{
// i is the index of the node to be max- heapified.
// size of heap is no. of nodes in heap.  (largest index of array)

int left = 2*i +1;
int right = 2*i +2;

int large = i;
if (left ≤ size && a[i] < a[left])
        large = left;
if (right ≤ size && a[large] < a[right])
        large = right;

if (large != i)
{ swap( a[i], a[large]);

  max-heapify (a, large, size);
}
}
```

* Max- heapification is used in both heap creation & heap deletion.

Date: _____ Page No: ____

(Algo ②)

```
void Build_Maxheap (int a[], int size)
{
    for (int i = (size-1)/2 ; i >= 0 ; i--)
        max_heapify (a, i, size);
}

void heapSort (int a[], int size)
{
    Build_Maxheap (a, size);
    for (i= size ; i>= 1 ; i--)
    {
        swap (a[0] , a[size]);
        size--;
        max_heapify ( a, 0, size);
    }
}

void main ( )
{
    int a[ ] = { 6, 12, 18, 3, 5, 16, 17 } ;
    heapSort ( a, 6) ;
    for (i=1 ; i<=n ; i++)
        printf ("%d", a[i]);
}
```



Date: _____ Page No: _____

Different Notations of Complexity : (Asymptotic Behaviour of Functions).

* Let, $f(n) = n^2 + 4$, $g(n) = 2^n$

- for $n = 1 \Rightarrow f(n) = 5, g(n) = 2 \Rightarrow f(n) > g(n)$
- for $n = 2 \Rightarrow f(n) = 8, g(n) = 4 \Rightarrow f(n) > g(n)$
- for $n = 3 \Rightarrow f(n) = 13, g(n) = 8 \Rightarrow f(n) > g(n)$
- for $n = 4 \Rightarrow f(n) = 20, g(n) = 16 \Rightarrow f(n) > g(n)$
- for $n = 5 \Rightarrow f(n) = 29, g(n) = 32 \Rightarrow f(n) < g(n)$

* growth rate of $g(n)$ is much more than that of $f(n)$.

* growth rate of exponential / factorial functions is much higher than that of polynomials.

(1) Big-Oh (O) :

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

growth rate of $g(n) \geq f(n)$

eg $f(n) = n^2 + 4$, $g(n) = 2^n$

$\Rightarrow$ $f(n) = O\ g(n)$ ✓

$g(n) = O\ f(n)$ ✗

eg $f(n) = 5n^2 + 4$, $g(n) = n^2$

$f(n) > g(n)$ ⇒ if highest term of n is same, growth rate will be the same.

→ $f(n) = O\ g(n)$ ✓
$g(n) = O\ f(n)$ ✓

* If $f(n) = O\ g(n)$ then $g(n) = O\ f(n)$ ⇒ False

• Thus, 'Big Oh' is not symmetric

• Mathematically Definition:

* $f(n) = O\ g(n)$ if $f(n) \leq C_1\ g(n)$; $n \geq n_0$, $C_1 > 0$

where, C_1 = +ve constant
n_0 = value of n after which this eq. is always satisfied

eg $f(n) = n^2 + 4$
$g(n) = 2^n$

for $C_1 = 1$ ⇒ $f(n) \leq 1 \cdot g(n)$

$n \geq 5$. $(n_0 = 5)$

for $C_1 = 1/2$ → $f(n) \leq 1/2\ g(n)$

$n \geq 7$ $(n_0 = 7)$

Date: Page No:

② Theta (Θ):

$f(n) = \Theta\, g(n)$ when;

growth rate of $f(n) = g(n)$

eg $f(n) = n^2 + 2$, $g(n) = 5n^2 + 6$

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

* If $f(n) = \Theta\, g(n)$ then $g(n) = \Theta\, f(n) \Rightarrow$ True

• Thus, 'Theta' is symmetric.

• Mathematical Definition:

* $f(n) = \Theta\, g(n)$ if $\forall n \geq n_0$

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

eg $f(n) = n^2 + 4$, $g(n) = 2n^2 + 6$

$c_1 = 1$, $c_2 = 2$ $\qquad f(n) = \Theta\, g(n)$

$\Rightarrow 1 \cdot f(n) \leq g(n) \leq 2 \cdot f(n) \qquad \forall n \geq 1 \quad (n_0 = 1)$

$(n^2+4) \leq 2n^2 + 6 \leq 2(n^2+4) \quad ; n \geq 1$

* If $f(n) = O\ g(n)$ then
$g(n) = \Omega\ f(n)$.

Date: Page No:

③ Big Omega (Ω):

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

growth rate of $g(n) \leq f(n)$

· Mathematical Definition:

* $f(n) = \Omega\ g(n)$ if $\forall n \geq n_0$

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

* If $f(n) = \Omega\ g(n)$ then $g(n) = \Omega\ f(n) \Rightarrow$ False

· Thus, 'Big Omega' is not symmetric.

* Transitive: If $f(n) = O\ g(n)$ & $g(n) = O\ h(n)$

then $f(n) = O\ h(n)$

* 'Big Oh', 'Theta' & 'Big Omega' are transitive.

* Reflexive: $f(n) = O\ f(n)$

* 'Big Oh', 'Theta' & 'Big Omega' are reflexive.

Date: Page No:

* Transpose Symmetricity: $f(n) = O\,g(n)$

if & only if $g(n) = \Omega\, f(n)$

Que Arrange the functions in ascending order of their growth rates.

⇒ 2^n, n^n, $\lfloor n$ ⇒ $2^n = 2 * 2 * 2 \_\_\_ n$

$n^n = n * n * n \_\_\_ n$

$\lfloor n = 1 * 2 * 3 \_\_\_ n$

Ans ⇒ $\lfloor n < 2^n < n^n$ ✗

$2^n < \lfloor n < n^n$ ✓

Que

⇒ $n^{1/3}$, $n^{7/5}$, $2^{\log_2 n}$ ⇒ $n^{1/3} = \sqrt[3]{n} = n^{0.33}$

$n^{7/5} = n^{1.4}$

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

Ans = $n^{0.33} < n^1 < n^{1.4}$

$n^{1/3} < 2^{\log_2 n} < n^{7/5}$

Que

⇒ 2^n, $n^{\log n}$

① Direct Observation
② Log
Best ③ Limits
④ Large values of n

It, sometimes gives incorrect answer. thus, it's not preferred.

* Before taking log, constants can be ignored but, after taking log, constants cannot be ignored.

Date: Page No:

② method

2^n, $n^{\log n}$ $\Rightarrow$ $2^n \Rightarrow \log_2 2^n = n \log_2 2 = n$

$n^{\log n} \Rightarrow \log_2 n^{\log_2 n} = (\log_2 n \log_2 n)$

$= (\log_2 n)^2$

Ans $n^{\log n} < 2^n$

Que $n^{0.1}$, $\log^9 n$ * $\log^9 n = (\log n)^9$

$\Rightarrow \log n^{0.1} = 0.1 * \log n$

$\Rightarrow \log \log^9 n = 9 \log(\log n)$

③ method

$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow g(n) > f(n)$

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

$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \text{constant} \Rightarrow f(n) = g(n)$

$\Rightarrow \lim_{n \to \infty} \frac{n^{0.1}}{\log^9 n}$ $\left(\frac{\infty}{\infty} \text{ form}\right)$ $\Rightarrow$ L'hospital's Rule

$\Rightarrow$ Diff. until any function becomes constant.

$n^{\epsilon} > \log^{k} n \quad ; \quad k, \epsilon > 0$

eg $n^2 > \log^{1000} n$

Date: _____ Page No: _____

$$\Rightarrow \lim_{n \to \infty} \frac{0.1\, n^{0.1-1}}{\frac{9 \log^8 n}{n}} = \frac{0.1\, n^{0.1-1} \cdot n}{9 \log^8 n} = \frac{0.1\, n^{0.1-1+1}}{9 \log^8 n}$$

$$\Rightarrow \lim_{n \to \infty} \frac{0.1\, n^{0.1}}{9 \log^8 n}$$

9 times differentiation

constant

$\therefore \log^9 n < n^{0.1}$

$2^n = \theta(2^{2n})$ ✗

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

$2^n = \theta(4^n)$

$(n+k)^m = \theta(n^m)$ ✓

* constants can be ignored if they are not in power.

$k, m \in$ constant.

$\log_2 n$ & $\log_{10} n$ have same growth rates.

But $n^{\log_2 n}$ & $n^{\log_{10} n}$ don't have same rates.

* $\theta(\log_2 n) = \theta(\log_{10} n)$ ✓ • Base of log can be ignored.

* $\theta(2^n) \neq \theta(2.00001)^n$ • Base of exponential can't be ignored.

$\Rightarrow 2^n = O(2.00001)^n$

Date: _____ Page No: ____

CS-2011 Que | IT-2008 Que 10 | CS-2008 Que 39 | CS-2015 (III) Que 29/35

* If $f(n) = \theta(g(n))$

then $2^{f(n)} = \theta(2^{g(n)})$ ⇒ False

eg $f(n) = n$, $g(n) = 2n$

$f(n) = \theta\, g(n)$ ⇒ $2^n = \theta(2^{2n})$

$2^n = \theta(4^n)$ ✗

---

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

then $f(\frac{n}{2}) = O\left(g(\frac{n}{2})\right)$ ⇒ ~~False~~ True

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

$3n = O(2n)$ ⇒ $\frac{3n}{2} = O(\frac{2n}{2})$ ✓

---

* $2^{n+1} = O(2^n)$ ✓

$2^n \cdot 2^1 = O(2^n)$

Date: Page No.

* $f(n) = \theta(f(\frac{n}{2}))$ ⇒ False

eg $f(n) = 3n \Rightarrow 3n = \theta(\frac{3n}{2})$ ✓

but, $f(n) = 4^n \Rightarrow 4^n = \theta(4^{n/2})$

$\Rightarrow 4^n = \theta(\sqrt{4})^n$

$\Rightarrow 4^n = \theta(2^n)$ ✗

* $O[f(n)] \cap \Omega[f(n)] = \theta[f(n)]$ ✓

* $O[f(n)]$ is a set of functions of complexity less than or equal to that of $f(n)$.
* $\Omega[f(n)]$ → greater than or equal to $f(n)$.
* $\theta[f(n)]$ → equal to $f(n)$.

* $f(n) + \Omega[f(n)] = O[f(n)]$ ✗

* $\log n \in O(n^2)$ ✓

* $\theta(f(n)) \subseteq O(f(n))$ ✓

Date: Page No:

* $f(n) = n^{\sin n}$, $g(n) = n$

(a) $f(n) = O[g(n)]$ X

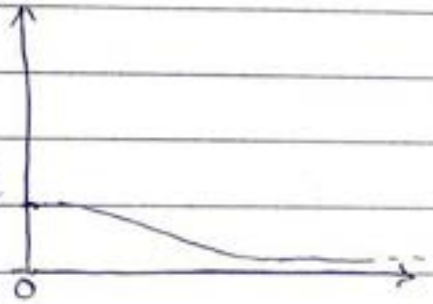
(b) $g(n) = O[f(n)]$ X

* $f(n)$ is a periodic function & these asymptotic notations are only for positive growing functions.

* $f(n) = 1/n$ ⇒ It's a decreasing function.

* But, for very large values of n, it will be constant.

$\therefore f(n) = O(1)$



But, $f(n) \neq \Theta(1)$

as $f(n) \neq \Omega(1)$

because $\frac{1}{n} \geq c_1 . 1$ can't be satisfied.

* Small Oh (o):

$f(n) = o[g(n)]$ if,

growth rate of $g(n) > f(n)$

Mathematically,

$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$ or $\lim_{n \to \infty} \frac{g(n)}{f(n)} = \infty$

Date: _____ Page No: _____

* Small Omega (ω):

$f(n) = \omega[g(n)]$ if

growth rate of $g(n) < f(n)$

Mathematically,

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

* $O(n^2)$ means growth rate can be equal or less than n^2.

* worst case → Upper Bound → Big Oh.
* Best case → lower Bound → Big Omega
* Average case → P Theta.

* Theta notation is used when complexity of algorithms is same in best, worst & avg. case.

eg selection sort ⇒ $\Theta(n^2)$, heap, merge ⇒ $\Theta(n \log n)$
but, Insertion ⇒ $\Theta(n^2)$ ✗

Also, worst case complexity of Insertion sort ⇒ $\Theta(n^2)$ ✓

Que Is $O(2^n)$ the complexity of Bubble Sort?

⇒ Yes ; $n^2 < 2^n$

Que Is $\Omega(\log n)$ the complexity of Bubble Sort?

⇒ Yes ; $n > \log n$

* $f(n) + g(n) = \Theta(\max(f(n), g(n)))$ ✓

* $f(n) + g(n) = \Omega(\min(f(n), g(n)))$ ✓

eg $n^2 + n = \Theta(\max(n^2, n)) = \Theta(n^2)$ ✓

eg $n^2 + n = \Omega(\min(n^2, n)) = \Omega n$ ✓

* If $f(n) = O(g(n))$ then $2^{f(n)} = O(2^{g(n)})$ ⇒ X

eg $f(n) = n^2$, $g(n) = n^3$ X

eg $f(n) = 2n$, $g(n) = 3n$ X

eg $f(n) = 3n$, $g(n) = 2n$ ✓

Date: _____ Page No: _____

* On the basis of complexity, algorithms can be;

① Sequential ② Iterative (loops) ③ Recursive (recursion)

↓

$\theta(1)$

⇒ How to compute complexity of iterative algorithms:

eg
```
for (i=1 ; i<=n; i++)
    sum = sum+i ;
```
⇒ $\theta(n)$

eg
```
for (i=1 ; i<=n ; i=i+2)
    sum = sum+i ;
```
⇒ $\theta\left(\frac{n}{2}\right) = \theta(n)$

eg
```
for (i=1 ; i<=n ; i= i*2)
    sum = sum+i ;
```
$n = 2^k \Rightarrow k = \log_2 n$

⇒ $\theta(\log_2 n + 1)$

⇒ $\theta(\log_2 n)$

eg
```
for (i=1 ; i<=n ; i= i*3)
    sum = sum+i ;
```
$n = 3^k \Rightarrow k = \log_3 n$ ⇒ $\theta(\log_3 n + 1)$

⇒ $\theta(\log_3 n)$

* Any loop which is going from 1 upto n & growing exponentially has complexity $\theta(\log n)$.
* Any loop which is decreasing exponentially from n to 1, then it'll have complexity $\theta(\log n)$.

eg `for (i=1; i<=2^n; i=i*2)`
`sum = sum + i;`
$\Rightarrow \theta(n+1)$
$\Rightarrow \theta(n)$

eg `for (i=1; i^2 <= n; i++)`
`sum = sum + i;`
$\Rightarrow$ loop is running i^2 times

$i^2 \le n$
$i \le \sqrt{n} \Rightarrow \theta(\sqrt{n})$

—x—

eg
```
i=1, k=1
while (k<=n)
{ i++;
  k= k+i; }
```

$\Rightarrow$ $i=1$, $k=1$ (let, $n=10$)
$i=2$, $k=1+2$
$i=3$, $k=1+2+3$
$i=4$, $k=1+2+3+4$
$i=5$, $k=1+2+3+4+5$

$\therefore \frac{i(1+i)}{2} \le n$

$\frac{i^2+i}{2} \le n$

or $i^2 \le n$

$i \le \sqrt{n} \Rightarrow \theta(\sqrt{n})$

Date _____ Page No. _____

eg
```
for (i=1 ; i<=n ; i++)        ---  n
                                    ⇒ O(n²)
    for (j=1 ; j<=n ; j++)    ----  n
        sum++;
    {
{
```

	inner loop runs
⇒ i=1	n times
2	n
3	n
!	!
n	n

⇒ n + n + n - - - n

⇒ n^2

⇒ $\Theta(n^2)$

eg
```
for (i=1 ; i<=n ; i++)
    for(j=1 ; j<=i ; j++)
        sum++;
    {
{
```

⇒ i=1	1
2	2
3	3
!	!
!	!
n	n

⇒ 1 + 2 + 3 - - - n

⇒ $\frac{n(n+1)}{2}$

⇒ $\Theta(n^2)$

$\log a + \log b + \log c = \log(a.b.c)$

Date: Page No:

eg
```
for (i=1 ; i<=n ; i++)        ---- n
{
    for(j=1 ; j<=n ; j=j*2)   ---- log n
    {
        sum++;
    }
}
```
$\Rightarrow \Theta(n \log n)$

eg
```
for (i=1 ; i<=n ; i++)
{
    for (j=1 ; j<=i ; j=j*2)
    {
        sum++;
    }
}
```
$\Rightarrow$ inner loop grows exponentially upto i.

$\therefore \Rightarrow (\log_2 i) + 1$

i = 1	$(\log_2 1) + 1$
2	$(\log_2 2) + 1$
3	$(\log_2 3) + 1$
!	!
!	!
!	!
n	$(\log_2 n) + 1$

Date: Page No:

$\Rightarrow \log_2 1 + \log_2 2 + \log_2 3 \quad \_\_\_\_ \quad \log_2 n$

$\Rightarrow \log_2 (1 * 2 * 3 \_\_\_\_ n)$

$\Rightarrow \log_2 (n!)$

But, for very large values of n, $n! \simeq n^n$

$\Rightarrow \log_2 (n^n) \Rightarrow n \log_2 n \Rightarrow \underline{\theta(n \log n)}$

eg
```
for (i=1 ; i<=n ; i= i*2)        ____ logn
{
   for (j=1 ; j<=n ; j++)        ____ n
   {
      sum++;                     ⇒ θ(nlogn)
   }
}
```

eg
```
for (i=1 ; i<=n ; i= i*2)
{
   for (j=1 ; j<=i ; j++)
   {
      sum++;
   }
}
```

⇒

i =		
1	(2^0)	1 (2^0)
2	(2^1)	2 (2^1)
4	(2^2)	4 (2^2)
!		!
!		!
!		!
n	(2^k)	n (2^k)

$\Rightarrow 2^0 + 2^1 \cdots 2^k$

$\Rightarrow \frac{2^0 (2^{k+1} - 1)}{2-1}$

$\Rightarrow 2^{k+1} - 1$

$\Rightarrow 2 \cdot 2^k - 1$

$\Rightarrow 2n - 1$

$\Rightarrow \underline{\theta(n)}$

OR → $2^{k+1} - 1 = 2^{\log_2 n + 1} - 1$

$= n^{\log_2 2} - 1 = \underline{\theta(n)}$

---

eg

```
for(i=1 ; i<=n; i++)
{
  for( j=1; j<=i^2 ; j++)
  {
    sum++;
  }
}
```

→

i =		
1		1 (1^2)
2		4 (2^2)
3		9 (3^2)
!		!
!		!
!		!
n		n^2

$\Rightarrow 1^2 + 2^2 \cdots n^2$

$\Rightarrow \frac{n(n+1)(2n+1)}{6}$

$\Rightarrow \underline{\theta(n^3)}$

Date: _____ Page No: ____

eg
```
sum = 0;
for (i=1; i<=n; i++)
{ sum = sum + i; }
```

Que The value of sum at the end of code is,

(a) $\Theta(n)$ (b) $\Theta(n^2)$

$\Rightarrow$ sum $= 1 + 2 + 3 + 4 \ldots n = \frac{n(n+1)}{2} \Rightarrow \underline{\Theta(n^2)}$

eg
```
sum = 0
for (i=1; i<=n; i++)
{
  for (j=1; j<=n; j++)
  {
    sum = sum + j;
  }
}
```

Que Find the value of sum in terms of complexity?

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

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

$\Rightarrow \underline{\Theta(n^3)}$

Date: _____ Page No: _____

eg
```
sum = 0
for( i=1 ; i<=n ; i++)
{
  for ( j=1 ; j<=n; j=j*2)
  {
    sum = sum+ j ;
  }
}
```

Find the value of sum?

⇒ $i=1$ → sum → $1 + 2 + 4 \_\_\_\_ n$

$(2^0 + 2^1 + 2^2 \_\_\_\_ 2^k)$

$2 \cdot 2^k - 1 = 2n-1$

1		
1		
n	sum ⇒	$n * (2n-1)$ ⇒ $\Theta(n^2)$

eg
```
for ( i=1 ; i<=n ; i++)
{
  for( j=1 ; j<=n ; j=j+i)
  {
    sum++;
  }
}
```

Date: Page No:

$i=1$	n
2	$n/2$
3	$n/3$
1	1
1	1
n	$n/n=1$

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

$\Rightarrow n\left(1 + \frac{1}{2} + \frac{1}{3} \cdots \frac{1}{n}\right)$

$\Rightarrow n\log n \Rightarrow \underline{\theta(n\log n)}$

---

eg
```
for (i = 1; i<=n; i++)          ---- n
{
  for (j=1; j<=n; j++)          ---- n
  {
    for (k=1; k<=100; k++)      ---- 100 [O(1)]
    {
      sum++;
    }
  }
}
```

$\Rightarrow \underline{\theta(n^2)}$

---

eg
```
for (i=1; i<=n; i++)
{
  for(j=1; j<=i^2; j++)
  {
    for (k=1; k<=(n/2); k++)
    {
      sum++;
    }
  }
}
```

		j	k
⇒	i = 1	1 (1^2)	n/2
	2	4 (2^2)	n/2
	3	9 (3^2)	n/2
	1	1	1
	1	1	1
	n	n^2	n/2

$$\Rightarrow 1.\frac{n}{2} + 2^2.\frac{n}{2} + \cdots\cdots n^2.\frac{n}{2}$$

$$\Rightarrow \frac{n}{2}\left(1^2 + 2^2 \cdots n^2\right)$$

$$\Rightarrow \frac{n}{2}\left(\frac{n(n+1)(2n+1)}{6}\right) \Rightarrow \underline{\theta(n^4)}$$

→

eg. for (i = n/2 ; i <= n ; i++) ---- n

  for (j = 1 ; j <= n ; j = j * 2) ---- log n

    for (k = 1 ; k <= n ; k = k * 2) ---- log n

      sum++; ⇒ $\underline{\theta(n \log^2 n)}$

    {

{{

→

Date: _____ Page No. _____

eg
```
for (i=1 ; i<=n ; i++)  ---- n
{
    for (j=1 ; j<=n ; j++)  ---- n
    sum = sum+j ; {
    for (k=1 ; k<=n ; k=k*2)  ---- logn
    sum = sum+k ; {
}
```

$\Rightarrow n * (n + \log n) \Rightarrow n^2 + n\log n \Rightarrow \underline{\theta(n^2)}$

e
```
for (i=2 ; i<=n ; i++)
{
    if (x % i == 0)
        break;
}
```

* If x is a prime no, loop will run n times.

∴ Best Case $\Rightarrow \theta(1)$

Worst Case $\Rightarrow \theta(n)$

e If an algorithm takes 4 seconds for n = 2 then what will be the time requirement for n = 16 if the complexity is;

(a) n	(b) nlogn	(c) n^2
$n=2 \Rightarrow 4$ sec	$\Rightarrow n = 2\log_2 2 \Rightarrow 4$ sec	$\Rightarrow n = 2^2 \Rightarrow 4$
$n=16 = \frac{4 \times 16}{2}$	$n = 16\log_2 16 \Rightarrow \frac{4 * 16\log_2 16}{2\log_2 2}$	$n = 16^2 = \frac{4 \times 16}{2^2}$
$\Rightarrow \underline{32 \text{ sec}}$	$\Rightarrow \underline{128 \text{ sec}}$	$\Rightarrow \underline{256 \text{ sec}}$

Algorithms which have complexity in exponential & factorials are intractable. (system cannot solve them in finite amount of time).

Date: Page No:

Que An algo takes 1 ns for n=1, then for n=64 how much time it'll take if complexity is 2^n.

⇒ for $n=1$; 2^1 ⇒ 1 ns

2^{64} ⇒ $\frac{1 \text{ ns} * 2^{64}}{2^1}$

⇒ 2^{63} ns ≈ 292.5 years

---

* To write a recursive code, first write recursive definition & then base or terminating condition.

eg Factorial ⇒ • Rec. defi : fact(n) = n * fact(n-1)

• base condi : fact(0) = 1

⇒
```
int fact (int n)
{
    if (n==0)
    return 1;
    else
    return n * fact(n-1);
}
```

Date: _____ Page No: ____

eg sum of first n natural nos.:

⇒ sum(n) = n + sum(n-1) & sum(0) = 0

eg fibonacci series : 0 1 1 2 3 5 8 13 ---

⇒ fib(n) = fib(n-1) + fib(n-2) & fib(0) = 0, fib(1) = 1

⇒
```
int fib (int n)
{
    if (n==0) return 0;
    if (n==1) return 1;
    return fib(n-1) + fib(n-2);
}
```

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

⇒ How to compute complexity of recursive programs:

* For this, we've to write recurrence relation.

eg
```
void P (int n)
{
    if (n<=1)
    return;
    else
    {
        printf ("%d", n);
        P(n-1);
    }
}
```

· Recurrence Relation ;

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

eg
```
void P (int n)
{
    if (n<=1)
        return;
    else
    {
        printf ("%d", n);
        P(n-1);
        P(n-1);
    }
}
```

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

* terminating condition;

$T(1) = 1$
(constant time)

---

eg
```
int P (int n)
{
    if (n<=1)
        return 1;
    else
        return n + P(n/2);
}
```

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

---

eg
```
int P (int n)
{
    if (n<=1)
        return 1;
    else
        return n + P(n/2) + P(n/2);
}
```

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

Date: _____ Page No: _____

eg
```
int P (int n)
{ if (n<=1)
    return 1;
  else
{ for (i=1; i<=n; i++)
  { sum = sum+i; }
  return sum + P(n-2);
}
{
```

$\Rightarrow T(n) = T(n-2) + n$

eg
```
int P (int n)
{ if (n<=1)
    return 1;
  else
  return n + 2 * P(n/2);
}
```

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

eg
```
int P (int n)
{ if (n<=1)
    return 1;
  else
  return n + P(n/2) * P(n/2);
}
```

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

* Iterative method gives exact answer eg. $\frac{n(n+1)}{2}$
while, master's method gives complexity eg. $O(n^2)$

Date / Page No

⇒ How to solve Recurrence Relation:

① Iterative / Substitution Method
★ ② Master's Method
③ Tree Method.

① Iterative Method:

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

* The recurrence relation & terminating condition is given, find the complexity of the algorithm.

⇒ $T(n) = T(n-1) + 1$

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

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

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

$\therefore T(n) = T(n-3) + 1 + 1 + 1$

$\therefore T(n) = T(n-k) + k$

Date: Page No:

Putting $n-k = 1$ as $T(1) = 1$

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

$T(n) = 1 + k$

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

$T(n) = n$

$\Rightarrow \underline{\Theta(n)}$

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

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

$= T\left(\frac{n}{4}\right) + 1$

$\therefore T(n) = T\left(\frac{n}{4}\right) + 1 + 1$

$T\left(\frac{n}{4}\right) = T\left(\frac{n}{8}\right) + 1$ $\quad \therefore T(n) = T\left(\frac{n}{8}\right) + 1 + 1 + 1$

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

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

$\Rightarrow k = \log_2 n$

$T(n) = T(1) + \log_2 n$

$T(n) = 1 + \log n$

$\Rightarrow \underline{\Theta(\log n)}$

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

Find (a) $T(7)$ (b) $T(100)$

(a) $T(1) = 1$

$T(2) = T(1) + 2 = 1 + 2 = 3$ $T(3) = 3 + 3 = 6$

$T(4) = 6 + 4 = 10$ $T(5) = 10 + 5 = 15$

$T(6) = 15 + 6 = 21$ $T(7) = 21 + 7 = 28$ Ans

(b) $T(100) = ?$

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

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

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

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

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

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

--- putting $n - k = 1 \Rightarrow k = n-1$

Date: Page No

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

$T(n) = \frac{n(n+1)}{2} \Rightarrow \underline{\underline{\Theta(n^2)}}$

$\therefore \quad T(100) = \frac{100(101)}{2} = \underline{\underline{5050}}$

→

eg $T(n) = 2T(n-1) + 1 \quad , \quad T(1) = 1$

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

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

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

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

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

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

Putting $n-k = 1 \Rightarrow k = n-1$

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

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

Date: Page No:

$\Rightarrow \frac{1.(2^n-1)}{2-1}$ $\qquad \frac{a(r^n-1)}{r-1}$

$\Rightarrow 2^n - 1$

Now find $T(6)$ & its complexity

- $T(6) = 2^6 - 1 = 64 - 1 = \underline{63}$
- complexity $\rightarrow \underline{\theta(2^n)}$

Date:_____ Page No:____

(2) Master's Method :

For $T(n) = a\,T\left(\frac{n}{b}\right) + f(n)\log^p n$

where, $a \geq 1$, $b > 1$, $p \geq 0$

eg $T(n) = T\left(\frac{n}{2}\right) + \frac{n^2}{\log n}$ ⇒ master's method is not applicable as $\log^{-1} n$ ($p = -1 \not\geq 0$)

eg $T(n) = 2T\left(\frac{4n}{3}\right) + n^2$ ⇒ not applicable as $b = 3/4 \neq 1$

$T(n) = 2T\left(\frac{n}{3/4}\right) + n^2$

→

⇒ compute $n^{\log_b a}$ & compare it with $f(n)$.

eg $T(n) = 4T\left(\frac{n}{2}\right)$ ⇒ $n^{\log_b a} = n^{\log_2 2^2} = \underline{n^2}$

eg $T(n) = 2T\left(\frac{n}{2}\right) + n$ ⇒ $n^{\log_2 2} = \underline{n}$

eg $T(n) = 2T\left(\frac{n}{4}\right) + n$ ⇒ $n^{\log_4 2} =$ ////

$\log_4 2 = \frac{1}{\log_2 4} = 1/2$

⇒ $n^{1/2} = \underline{\sqrt{n}}$

- $\boxed{\log_b a = \frac{1}{\log_a b}}$

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

$\Rightarrow n^{\log_2 4} = n^2$, $\Rightarrow f(n) = n$

Case (1)

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

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

$n^2 > n$

$\Rightarrow \Theta(n^2)$

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

$\Rightarrow n^{\log_2 2} = n$, $\Rightarrow f(n) = n$

Case (2)

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

$T(n) = \Theta(f(n) \cdot \log^{p+1} n)$

$n = n$

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

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

$\Rightarrow n^{\log_2 2} = n$, $f(n) = n$

$n = n$

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

Date: _____ Page No: ____

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

$\Rightarrow n^{\log_2 2} = n$, $f(n) = n^2$

Case (3) $f(n) = \omega(n^{\log_b a})$ OR

$n^{\log_b a} = o(f(n))$	$n < n^2$
$T(n) = \theta(f(n) \cdot \log^p n)$	$\Rightarrow \theta(n^2 \log^* n)$

—x—

Que $T(n) = T\left(\frac{n}{2}\right) + 1 \Rightarrow n^{\log_2 1} = n^0 = 1$

$f(n) = 1 \Rightarrow \theta(1 \cdot \log^{0+1} n)$

$T(n) \Rightarrow \theta(\log n)$

—x—

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

$\Rightarrow n^{\log_2 2} = n$, $f(n) = 1$

$\Rightarrow n > 1 \Rightarrow \Rightarrow \theta(n)$

—x—

Que $T(n) = 2T\left(\frac{n}{2}\right) + n \Rightarrow n^{\log_2 2} = n$, $f(n) = n$

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

—x—

Que $T(n) = 8T\left(\frac{n}{2}\right) + n^2 \log n \Rightarrow n^{\log_2 8} = n^3$

$n^3 > n^2 \Rightarrow \theta(n^3)$

—x—

Date: _____ Page No: _____

* Now, $p < 0$ (p can be negative also)

$T(n) = aT\left(\frac{n}{b}\right) + f(n) \log^p n \quad ; \quad a \geq 1, \; b > 1$

case (1) : $n^{\log_b a} = \omega(f(n))$

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

case (2) : $n^{\log_b a} = \Theta(f(n))$

If $p > -1 \Rightarrow T(n) = \Theta(f(n) \cdot \log^{p+1} n)$

If $p = -1 \Rightarrow T(n) = \Theta(f(n) \cdot \log(\log n))$

If $p < -1 \Rightarrow T(n) = \Theta(f(n))$

case (3) $n^{\log_b a} = o(f(n))$

If $p \geq 0 \Rightarrow T(n) = \Theta(f(n) \cdot \log^p n)$

If $p < 0 \Rightarrow T(n) = \Theta(f(n))$

Date: Page No:

Que $T(n) = T\left(\frac{n}{2}\right) + \frac{1}{\log n}$ $\Rightarrow$ $n^{\log_2 1} = n^0 = 1$

$f(n) = 1$

$\Rightarrow$ $1 = 1$ & $p = -1$

Ans $\Rightarrow$ $\Theta(1 \cdot \log(\log n))$

Que $T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{\log n}$ $\Rightarrow$ $n^{\log_2 2} = n$

$\Rightarrow$ $n = n$ & $p = -1$ $f(n) = n$

Ans $\Rightarrow$ $\Theta(n \cdot \log(\log n))$

Que $T(n) = 2T\left(\frac{n}{2}\right) + \frac{1}{\log^2 n}$ $\Rightarrow$ $n^{\log_2 2} = n$

$f(n) = 1$

$n > 1$ & $p = -2$

Ans $\Rightarrow$ $\Theta(n)$

Que $T(n) = 2T\left(\frac{n}{4}\right) + \frac{\sqrt{n}}{\log^2 n}$ $\Rightarrow$ $n^{\log_4 2} = n^{1/2} = \sqrt{n}$

$f(n) = \sqrt{n}$

$\sqrt{n} = \sqrt{n}$ & $p = -2 < -1$

Ans $\Rightarrow$ $\Theta(\sqrt{n})$

Date: _____ Page No: _____

* For $T(n) = aT(n-b) + n^k$

Case ① ⇒ If $a < 1$ ⇒ $T(n) = O(n^k)$ (Not $\Theta(n^k)$) *

Case ② ⇒ If $a = 1$ ⇒ $T(n) = O(n^{k+1})$

Case ③ ⇒ If $a > 1$ ⇒ $T(n) = O(n^k \cdot a^{n/b})$

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

⇒ $a = 1$ ⇒ $O(n^{0+1})$ ⇒ $O(n)$

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

⇒ $a = 1$ ⇒ $O(n^2)$

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

⇒ $a = 2 > 1$ ⇒ $O(n^0 \cdot a^{n/1})$ ⇒ $O(a^n)$ ⇒ $O(2^n)$

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

⇒ $a = 2 > 1$ ⇒ $O(n^1 \cdot 2^{n/1})$ ⇒ $O(n \cdot 2^n)$

Date: Page No:

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

assume $n = 2^m$

$T(2^m) = T(2^{m/2}) + 1$

let, $S(m) = S(m/2) + 1$

Now, master's method can be applied.

$\Rightarrow m^{\log_b a} = m^{\log_2 1} = m^0 = 1$

$f(m) = 1 \quad \Rightarrow 1 = 1 \quad \Rightarrow S(m) = \Theta(1 \cdot \log m)$

$\because m = \log_2 n$

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

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

$\Rightarrow n = 2^m$

$T(2^m) = 2T(2^{m/2}) + 1$

let, $S(m) = 2S(m/2) + 1$

$\Rightarrow m^{\log_2 2} = m$ & $f(n) = 1$

$\Rightarrow m > 1 \Rightarrow S(m) = \theta(m)$

$\because m = \log_2 n \quad \therefore = \theta(\log n)$

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

$T(2^m) = 2T(2^{m/2}) + 2^m$

$S(m) = 2S(m/2) + 2^m$

$\Rightarrow m^{\log_2 2} = m$ & $f(n) = 2^m$

$m < 2^m \Rightarrow \theta(2^m \cdot \log^0 n) \Rightarrow \theta(2^m)$

$\rightarrow \because 2^m = n \Rightarrow \underline{\theta(n)}$

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

$\Rightarrow \frac{T(n)}{n} = \frac{\sqrt{n}\, T(\sqrt{n}) + n}{n}$

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

$\frac{T(2^m)}{2^m} = \frac{T(2^{m/2})}{2^{m/2}} + 1$

$S(m) = S(m/2) + 1$

$m^{\log_b a} = m^{\log_2 1} = m^0 = 1$

$S(m) = \theta(\log m)$

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

Date: Page No:

G.S. 2004

Que 84 $T(n) = 2T(n-1) + n$, $T(1) = 1$ evaluates to;

(a) $2^{n+1} - n - 2$ (b) $2^n - n$

(c) $2^{n+1} - 2n - 2$ (d) $2^n + n$

* For evaluation or exact answer, master's method can't be used as it gives complexity not exact answer. Thus, option elimination can be used.

⇒ ∵ $T(1) = 1$ thus, put $n=1$ in options.

(a) $2^2 - 1 - 2 = 1$ (b) $2^1 - 1 = 1$

(c) $2^2 - 2 - 2 = 0$ (d) $2^1 + 1 = 3$

⇒ put $n=2$ ⇒ $T(2) = 2T(1) + 2 = 2 + 2 = 4$

(a) $2^3 - 2 - 2 = 4$ (b) $2^2 * 2 = 8$

G.S. 2005

Que 37 $T(n) = 2T\left(\frac{n}{2}\right) + n$, $T(1) = 1$

which is FALSE?

(a) $T(n) = O(n^2)$ (b) $T(n) = \Theta(n \log n)$

(c) $T(n) = \Omega(n^2)$ (d) $T(n) = O(n \log n)$

$T(n) = \Theta(n \log n)$ (by master's method)

CS-2014(I)

Que

Que Find the complexity.

input size	time (in sec)
10	10
20	19
30	29
40	41

$\Rightarrow O(n)$

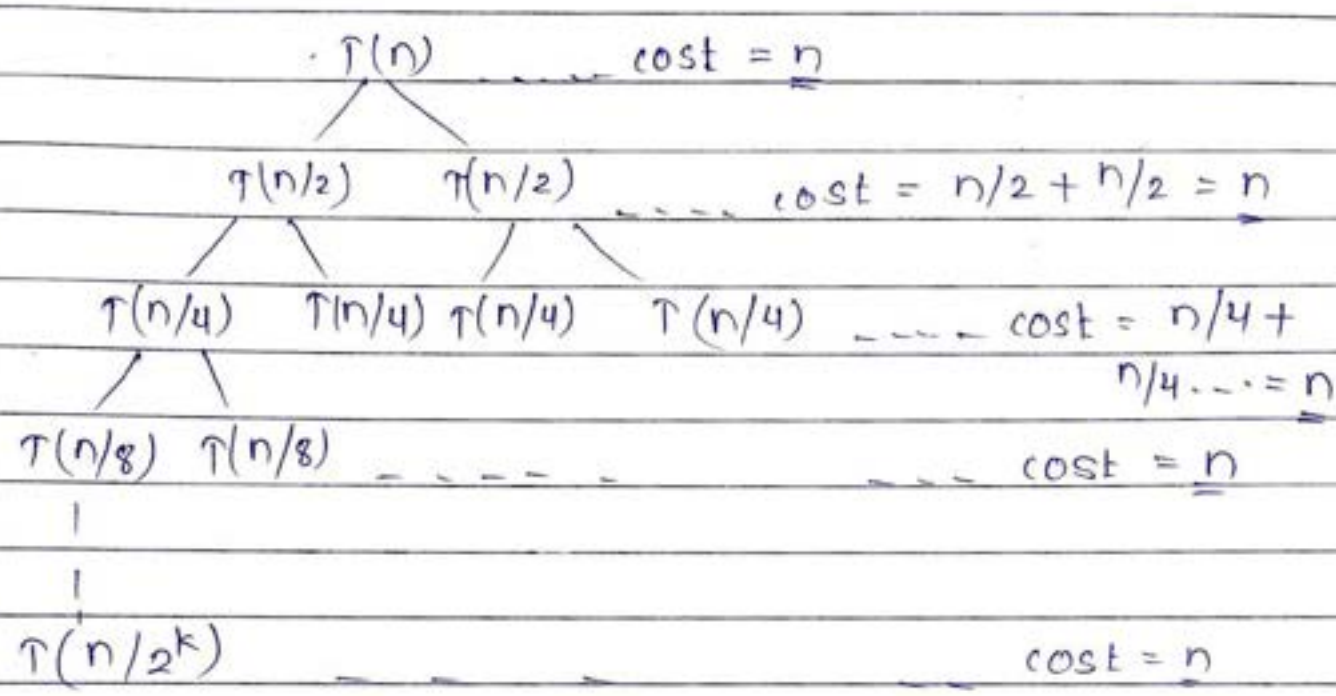
Date:_____ Page No.:_____

③ Tree Method :

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

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

↳ time/cost of n^{th} recursion



⇒ putting $(n/2^k) = 1$ as $T(1) = 1$

⇒ $n = 2^k$ ⇒ $k = \log_2 n$

no. of levels = $k + 1$ & cost of each level = n

⇒ sum of cost of every level will be the complexity of recurrence relation.

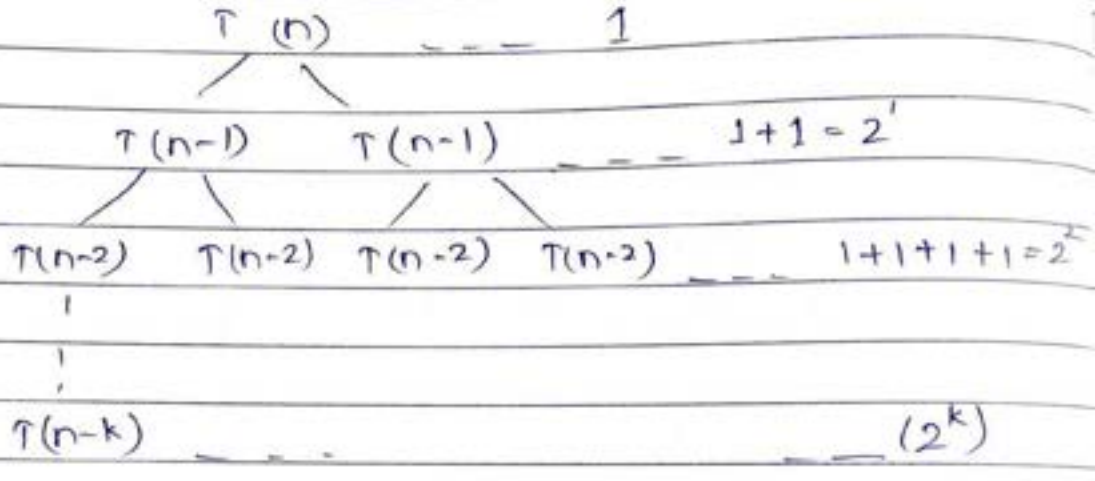
⇒ $(k+1) * n$ ⇒ $(\log_2 n + 1)\, n = n\log_2 n + n$ (exact answer)

⇒ $\Theta(n \log n)$

Date: Page No:

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

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



putting $n-k = 1 \Rightarrow k = n-1$

$\Rightarrow 1 + 2^1 + 2^2 \dots 2^k = 1 \cdot \left(\frac{2^{k+1} - 1}{2-1}\right) = 2^{k+1} - 1$

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

$\Rightarrow \underline{\theta(2^n)}$

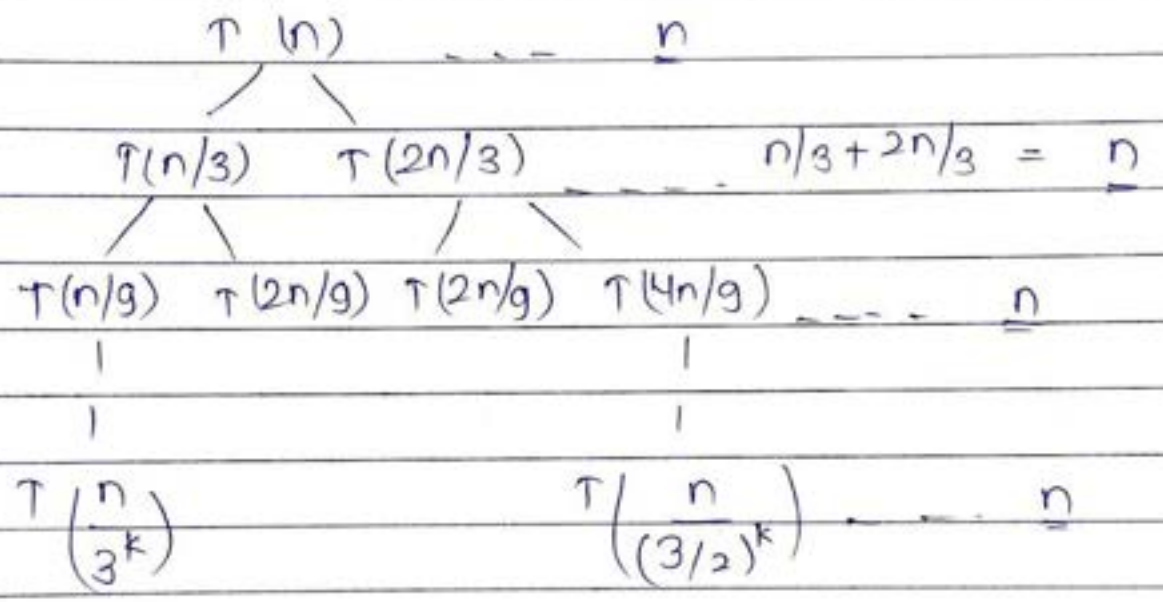
* There are some questions which have to be solved only by tree method ↓↓↓

Date:_____ Page No:_____

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

$T\left(\frac{n}{3}\right) = T\left(\frac{n}{9}\right) + T\left(\frac{2n}{9}\right) + n/3$

$T\left(\frac{2n}{3}\right) = T\left(\frac{2n}{9}\right) + T\left(\frac{4n}{9}\right) + 2n/3$



$\because T\left(\frac{n}{(3/2)^k}\right) > T\left(\frac{n}{3^k}\right)$ $\therefore$ putting $\frac{n}{(3/2)^k} = 1$

$\Rightarrow n = \left(\frac{3}{2}\right)^k \Rightarrow k = \log_{3/2} n$

$\Rightarrow$ no. of levels $\Rightarrow k+1$ & cost $= n$

$\Rightarrow n * (k+1) \Rightarrow n\left[(\log_{3/2} n)+1\right] \Rightarrow \underline{\Theta(n \log n)}$ (not exact)

Date: Page No:

Trick

$f(n) = \theta(n)$

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

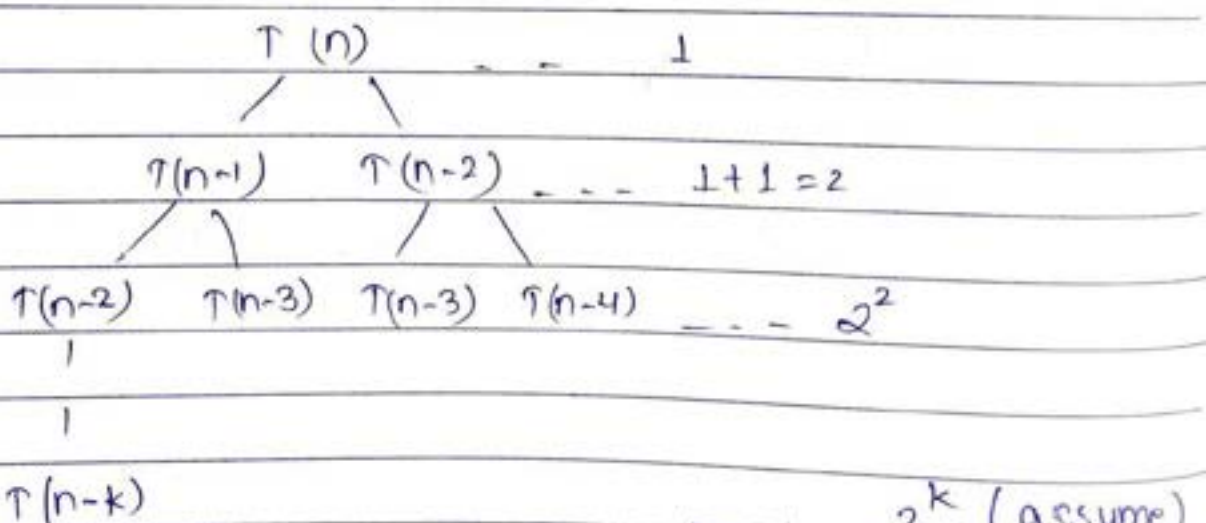
Here, $\frac{n}{3} + \frac{2n}{3} = \theta(n)$ = n ← then complexity will always be $(n \log n$

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

$\Rightarrow \frac{n}{4} + \frac{3n}{4} = \theta(n)$ = n → $O(n \log n)$

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

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



putting $n-k = 1 \Rightarrow k = n-1$

$\Rightarrow 1 + 2 + 2^2 \ldots 2^k = \frac{1 \cdot (2^{k+1} - 1)}{(2-1)} = 2^{n-1+1} - 1$

$\Rightarrow \theta(2^n)$

(not exact)

Date:_____ Page No:_____

(5) Quick Sort: (Divide & Conquer Sort)

* we select an element as pivot element (generally the left most element)

eg 10 5 15 6 12 9 13 7 8 18

* In right to left scan, we search for the element smaller than pivot. & in left to right scan, we search for greater one.

⇒ 10 5 15 6 12 9 13 7 8 18
pivot low L H high

* Now, swap the elements pointed by high & low.

⇒ 10 5 8 6 12 9 13 7 15 18

* After swapping, increment low & decrement high.

⇒ 10 5 8 6 12 9 13 7 15 18
L L H

⇒ 10 5 8 6 7 9 13 12 15 18
L H
H L

Date: _____ Page No: _____

* When low crosses high, stop partition procedure. & swap high & pivot.

⇒ 9 5 8 6 7 10 13 12 15 18

* Now, pivot is at its position
* Now do the same for all further sublists unless each sublist is of only one element.

Que (20) 10 25 8 30 19 22 2 26
P L H H

⇒ 20 10 2 8 30 19 22 25 26
P L H H

⇒ 20 10 2 8 19 30 22 25 26
P H L

→ 19 10 2 8 | 20 | 30 22 25 26
P1 L1 L1 H1 P2 L2 L2 H2
L1 L2

⇒ | 8 | 10 2 | 19 | 20 | 26 22 25 | 30
P1 L1 H1 P2 L2 H2
L2

⇒ 8 | 2 10 | 19 | 20 | 25 22 | 26 | 30
H1 L1

⇒ 2 | 8 | 10 | 19 | 20 | 25 22 | 26 | 30
P H L

⇒ 2 8 10 19 20 22 25 26 30

Best case : when pivot divides the list in 2 equal parts

worst case : when pivot doesn't divide the list.

Date: Page No:

* Complexity of Quick Sort :

eg

1	2	3	4	5	6
P	$\uparrow_L$	$\uparrow_H$	$\uparrow_H$	$\uparrow_H$	$\uparrow_H$
$\uparrow_H$	$\uparrow_H$				

comparisons ⇒ (6-1), (5-1), (4-1), (3-1)(2-1)

* There are (n-1) comparisons in quick sort in pass (1) worst case. (when array is already sorted)

* list is not divided into 2 sublists, in worst case.

⇒

(1)	2	3	4	5	6
P	$\uparrow_L$	$\uparrow_H$	$\uparrow_H$	$\uparrow_H$	$\uparrow_H$
$\uparrow_H$	$\uparrow_H$				

(n-2) comparisons in pass (2)

⇒ $(n-1)+(n-2)+\ldots 2+1$

⇒ $\frac{n(n-1)}{2}$ ⇒ $O(n^2)$.

* when list is divided in nearly equal no. of elements ⇒ Best case of Quick sort.

⇒ $T(n) = 2T\left(\frac{n}{2}\right) + \underbrace{(n-1)}_{\text{comparisons}}$

⇒ $O(n \log n)$

* If median is selected as pivot, list will definitely be divided in 2 equal parts. Date: Page No:

eg 8 6 | 9 | 20 30 21 25

n/3 p 2n/3

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

$\Rightarrow O(n \log n)$

* If all elements are equal.

eg 3 3 3 3 3 3

P L H H H H

H H

$\Rightarrow$ 3 | 3 3 3 3 3

* list is not divided $\Rightarrow$ worst case $\Rightarrow O(n^2)$

CS. 2014(1) Que 14 CS. 2014(3) Que 14 CS. 2015(1) Que 14

* there exists an algo. of complexity $O(n)$ that can find median of the list. without sorting.

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

$\Rightarrow O(n \log n)$ (same complexity)

* q contains pivot
* complexity of partition function is (n-1) Date: Page No:

Algorithm of Quick sort :

```
int partition ( int a[], int left, int right )
{
    int pivot = a[left];
    int low = left + 1;
    int high = right
    while ( low <= high)
    {
      while (a[high] >= pivot)
       { high--; }
      while ( a[low] <= pivot  && low <= high)
      { low++; }
      if (low <= high)
    {
      swap ( a[low], a[high]);
       low++ ; high--;
    }
    swap(a[left], a[high]);
    return high;
    }

void quicksort ( int a[], int left, int right)
{
   if (left < right)
  { int q = partition (a, left, right);
    quicksort (a, left, q-1);
    quicksort (a, q+1, right);
} }
```

—x—

Date: Page No:

⑥ Merge Sort : (Divide & Conquer sort)

* We half the list unless each sublist has only one element.

eg | 12 | 8 | 15 | 9 | 16 | 7 | 1 | 10 |

* Now, we'll merge the sublists & sort them by putting the elements one by one in an extra arra

⇒ 8 12 | 9 15 | 7 16 | 1 10

⇒ 8 9 12 15 | 1 7 10 16

⇒ 1 7 8 9 10 12 15 16

* Complexity of Merge Sort :

* The complexity of merging the sublists at every level is O(n) because all the elements have to be shifted from additional array to original array.

eg 10 16 18 21 | 100 105 106 110

⇒ 10 16 18 21 100 105 106 110

* Though, comparisons are only 4 [(10-100), (16-100), (18-100), (21-100)], but all elements have to be shifted in main array after sorting.

there are (k+1) levels but at level k (topmost level) no need of merging, thus, ⇒ (k+0) * n ✓ (not (k+1)*n) ✗

Date: Page No:

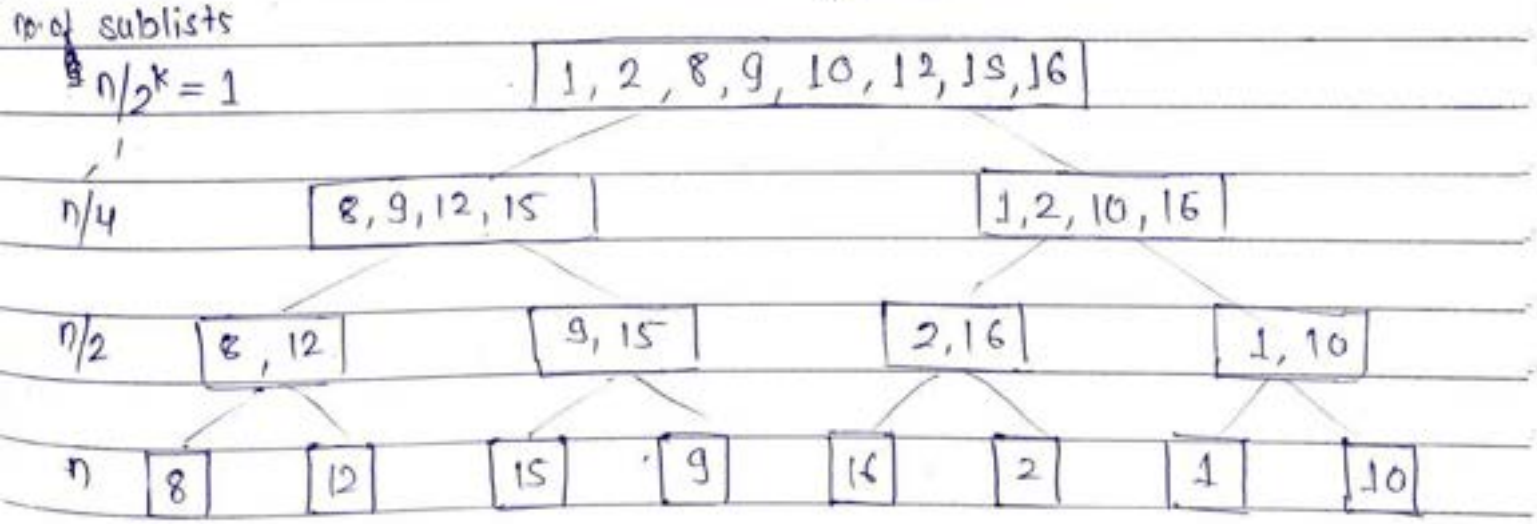
Thus ⇒ max. no. of comparisons in merging + complexity of shifting all the elements in main array

⇒ (n-1) + n ⇒ O(n).

eg 5 7 9 | 6 8 10

→ 5 6 7 8 9 10

comparisons ⇒ (5-6), (6-7), (7-8), (8-9), (9-10) ⇒ 5 i.e. (n-1) for n elements



* For 2^k elements, there will be (k+1) levels.
* Complexity of merging at each level is O(n).

⇒ k * n & $2^k = n$ ⇒ $k = \log_2 n$

∴ ⇒ O(n log n)

Date: Page No:

Que 10 6 15 12 3 1 7 2

can it be the status of array in merge sort?

(1) 6 10 12 15 | 3 1 7 2
✓ ✗

⇒ Yes, merge sort algo. sorts first sublist completely & then sorts second sublist.

(2) 10 6 12 15 3 1 7 2

⇒ 10 6 12 15 | 3 1 7 2
✗ ✗

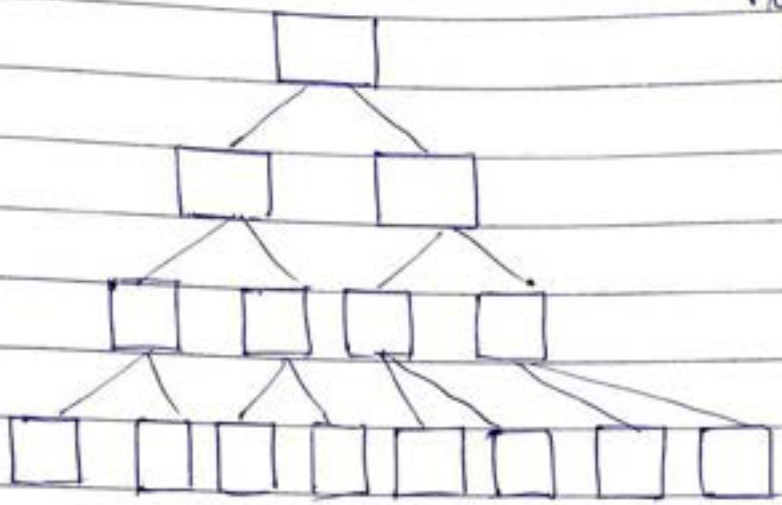
⇒ 10 6 | 12 15 | 3 1 | 7 2
✗ S1 ✓ S2 ✗ ✗

⇒ No, merge sort algo. starts sorting the sublists from left to right.

Date: _____ Page No: ____

2005

Que we are having $\log n$ sorted lists of size $(n/\log n)$, what is the complexity to merge them in a single sorted list.



⇒ no. of elements ⇒ $\frac{n}{\log n} + \frac{n}{\log n}$ ---- $\log n$ times

⇒ $\frac{n}{\log n} * \log n$ ⇒ n elements

* complexity of merging n elements is $O(n)$.

⇒ there are k levels (or $k+1$ levels),

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

$\therefore k = \log \log n$

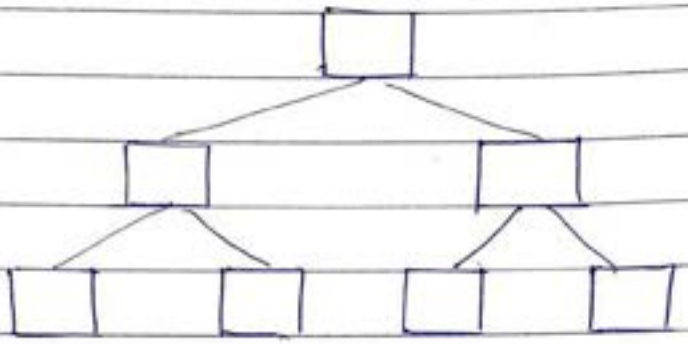
⇒ Complexity ⇒ complexity of merging * total levels

⇒ $n * \log \log n$ ⇒ $O(n \log \log n)$

Date: _____ Page No: ____

Que We are having 4 sorted sublists of size $n/4$, what is the complexity to merge them in single sorted list?

⇒



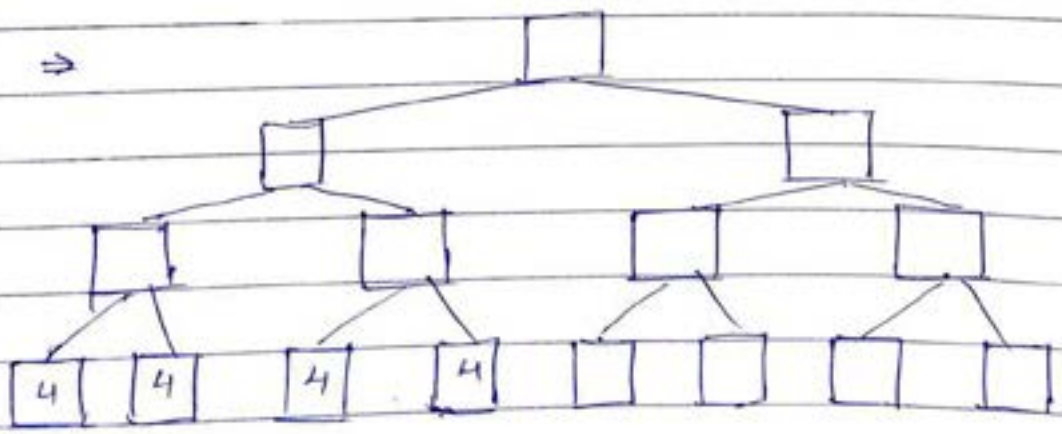
⇒ no. of elements ⇒ $\frac{n}{4} + \frac{n}{4} + \frac{n}{4} + \frac{n}{4} = \frac{4n}{4} = n$

∴ complexity of merging n elements is $O(n)$.

⇒ There are 3 levels ⇒ ∴ complexing $= n * 2$
$= 2n$
$\Rightarrow \underline{O(n)}$

Que We are having $n/4$ sorted sublists of size 4, what is the complexity to merge them in a single sorted list?

⇒



$n/2^k = 1$

$n/16$ $(n/2^4)$

$n/8$ $(n/2^3)$

... $n/4$ $(n/2^2)$

Date: Page No:

→ no. of elements ⇒ $4+4+\ldots\frac{n}{4}$ ⇒ n

* There are k levels (or k-1 levels),

⇒ $\frac{n}{2^k} = 1$ ⇒ $k = \log n$ ⇒ $n*k \Rightarrow O(n\log n)$

Que We are having list of n strings of size n, what is the complexity to sort them using merge sort.

⇒

Algorithm of Merge Sort :

eg

3	6	10	12	2	5	11	18
0	1	2	3	4	5	6	7
i				j			

- start = 0 ⇒ it points to start of sublist. ①
- end = 7 ⇒ it points to end of sublist ②
- mid = $\frac{0+7}{2} = 3$ ⇒ it points to end of sublist ①.
- mid + 1 = 4 ⇒ it points to start of sublist ②.

⇒ i = start, j = mid + 1

⇒
```
void merge (int a[], int start, int mid, int end)
{
    int i=start;
    int j= mid+1;
    int k= start;
    int b[end-start+1];        //additional array
    for (int p = start; p<=end; p++)
    {
        if (i > mid)  b[k++] = a[j++];
        else if (j > end)  b[k++] = a[i++];
        else if (a[i] < a[j])
            b[k++] = a[i++];
        else
            b[k++] = a[j++];
    }
```

* complexity of merging function is n.

Date:_____ Page No:_____

```
for (p = start ; p<= end ; p++)
{
    a[p] = b[p];
}
```

```
void mergesort (int a[], int start, int end)
{
    int mid;
    if (start < end)
    {
        mid = (start + end) / 2;
        mergesort (a, start, mid);
        mergesort (a, mid+1, end);
        merge (a, start, mid, end);
    }
}
```

→

* Complexity of dividing the list into sublists:

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

* Complexity of merging :

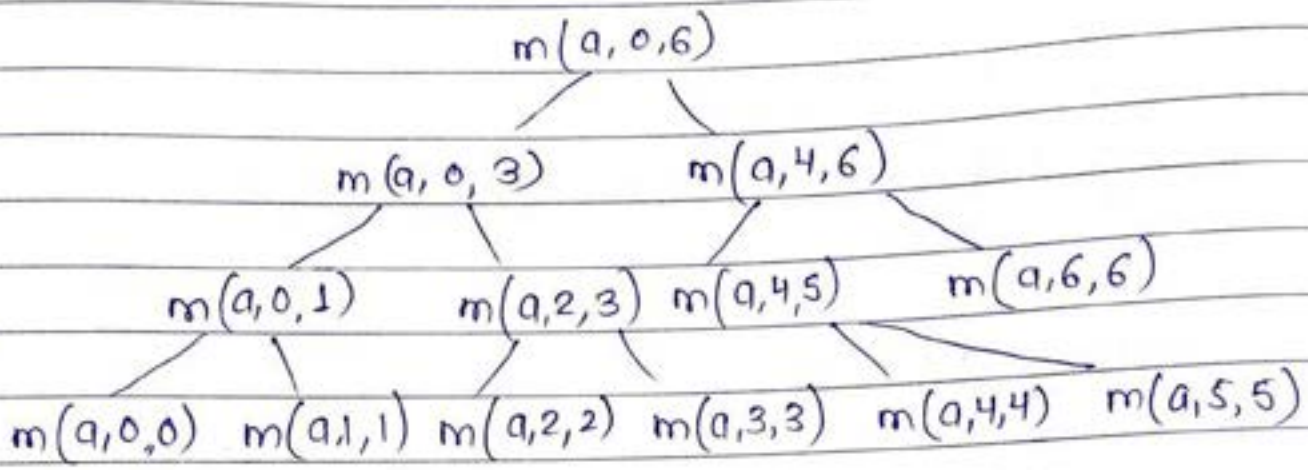
$$T(n) = 2T\left(\frac{n}{2}\right) + n \quad \Rightarrow \quad O(n \cdot \log n)$$

∴ Overall complexity = $n + n\log n \Rightarrow O(n\log n)$

Date: _____ Page No: _____

Que If mergesort (a, 0, 6) is called from main(), then how many recursive calls are made.?

⇒



⇒ 12 recursive calls.

Date: _____ Page No: _____

* Lower Bound Theorem:

"No comparison based sorting can perform better than $(n \cdot \log n)$ in worst case."

Que Recurrence relations of given sorting & searching algo:

① Selection Sort	(A) $T(n) = 2T(\frac{n}{2}) + n \Rightarrow O(n \log n)$
② Merge Sort	(B) $T(n) = T(n-1) + 1 \Rightarrow O(n)$
③ Linear Search	(C) $T(n) = T(n-1) + n \Rightarrow O(n^2)$

Counting Based Sorting

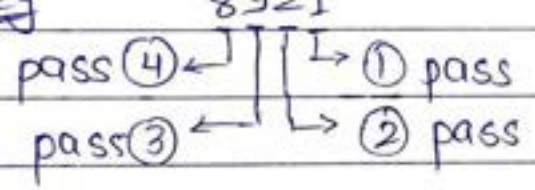
① Radix Sort : It is not inplace as it uses queues & no. of queues is equal to base or radix of the no. system (10 for decimal).

* We assume that the width of every no is equal. i.e.

8921 ⇒ 8921 , 1 ⇒ 0001

x No. of passes is equal to width (digits) of numbers.

eg



eg 65 72 89 51 19 63 10 47

	Pass ①	Pass ②
q_0	10	
q_1	51	10, 19
q_2	72	
q_3	63	
q_4		47
q_5	65	51
q_6		63, 65
q_7	47	72
q_8		89
q_9	89, 19	

⇒ 10, 19, 47, 51, 63, 65, 72, 89

* Radix sort gives better complexity than that of all comparison based sorting but yet it's not preferable as it's not inplace.

Date:_____ Page No:____

* Complexity of Radix Sort:

```
for (i=1; i<=w; i++)    // for width or passes (w)
{
  for (j=1; j<=n; j++)  // for total elements (n)
  {
    if (ith digit of a[j]==0)
      put a[j] in q0 {
    if (ith digit of a[j]==1)
      put a[j] in q1 {
  }
}
(for all queues)
```

⇒ no. of passes = width (w)

total elements = n

∴ complexity ⇒ $\Theta(nw)$

* If width (w) is constant, radix sort complexity:

⇒ $\Theta(n)$

Que can radix sort perform in O(n) in worst case

⇒ Yes, (when w = constant)

• $\log_8$ for octal, $\log_{16}$ for hexa, etc.

Date: Page No.

Que Size of array is n & if numbers are in the ran of 0 to n^2 & are represented in binary. What wi be the complexity of radix sort to sort this array.

⇒ let k be the no. of bits required to represent n^2 in binary.

eg $(12)_{10} = (1100)_2 \Rightarrow \log_2 12 = \lceil 3.58 \rceil = 4$ ✓

$(8)_{10} = (1000)_2 \Rightarrow \log_2 8 = \lceil 3 \rceil = 3$ ✗

$\therefore (8)_{10} = (1000)_2 \Rightarrow \lfloor \log_2 8 \rfloor + 1 = 4$ ✓

thus, the correct formula will be,

$\Rightarrow \lfloor \log_2 n \rfloor + 1$

$\therefore k = \lfloor \log_2 n^2 \rfloor + 1$

$k = \lfloor 2 \times \log_2 n \rfloor + 1 \approxeq \log n$

$\therefore$ complexity will be $\Rightarrow \Theta(n \cdot \log n)$

$\Rightarrow \Theta(n * \log n)$

$\Rightarrow \underline{\Theta(n \log n)}$ Ans

Date: Page No:

Q.12 The numbers are represented in binary & the range is from 0 to 2^n, the size of the list is n then what will be the complexity of radix sort?

$\Rightarrow \lceil \log_2 2^n \rceil + 1 = \lceil n \log_2 2 \rceil + 1 = \lceil n \rceil + 1 \cong n$

$\therefore$ complexity $\rightarrow \Theta(n.w) = \Theta(n.n) = \underline{\Theta(n^2)}$

* Thus, radix sort doesn't always perform in $O(n)$. It depends on the range of numbers.

If -10, to +10 ⇒ size of array = 20

Date: Page No:

② Counting Sort :

* It requires an array & its size will be the largest possible element. Thus, we must know the range of the no. in the very beginning.

eg 1 2 7 100 ⇒ size of array ⇒ 100

eg range ⇒ (1 to 9)

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

* let, index starts from 1 (not 0)

* take additional array with size equal to largest elemer (7 here)

b ⇒

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

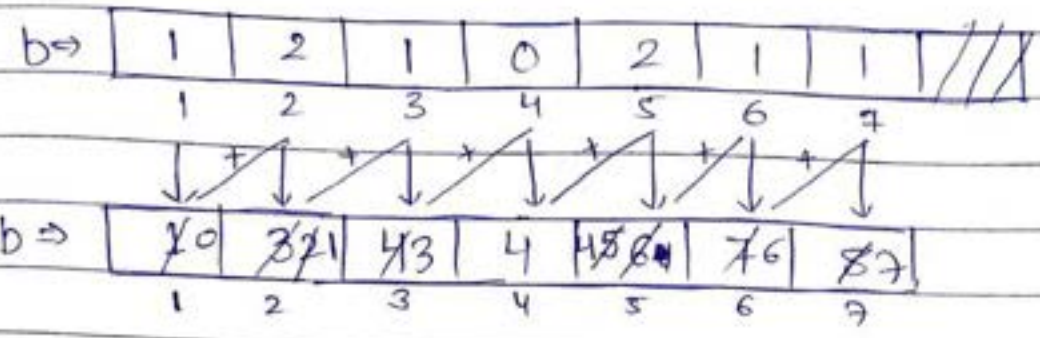
* initialize all indices of b[] to zero.

⇒ Now a[1] = 2 ⇒ increment b[2] by 1.

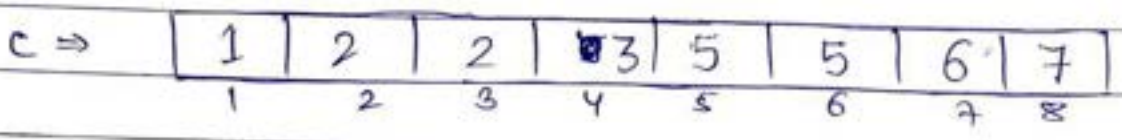
similarly, a[2] = 5 ⇒ inc. b[5] by 1

& so-on upto a[8].

Date: _____ Page No: ____



* Now, take another array c of size equal to that of array a.



⇒ a[1] = 2 ⇒ b[2] = 3
thus 2 ⇒ c[3] & dec. b[2]

Similarly a[2] = 5 ⇒ b[5] = 6
thus, 5 ⇒ c[6] & dec. b[5]

* If we start from a[10] not from a[1] it'll become stable.

* Complexity of Counting Sort:

⇒ O(n)

P.T.O.

Date: _____ Page No: _____

```
for (i = 0; i < n; i++)        // n
    b[i] = 0;
for (i = 0; i < n; i++)        // n
    b[a[i]]++;
for (i = 1; i < n; i++)        // n
    b[i] = b[i-1] + b[i];
for (i = 1; i < n; i++)        // n
    c[b[a[i]]] = a[i];
    b[a[i]]--;
}
```

⇒ n+n+n+n

O(n)

Date: Page No:

Searching :

① Linear Search :

eg 6 5 10 7 9 18 2 ; search 2

* (n-1) comparisons in worst case ⇒ O(n)

eg 2 6 5 10 7 9 18 ; search 2

* 1 comparison in best case ⇒ O(1)

* (n/2) comparisons in average case ⇒ O(n)

⇒
```
for (i=1; i<=n; i++)
{
  if (a[i] == x)
  {
    printf ("found"); break;
  }
}
if (i>n)
printf ("Not found");
```

Date: Page No:

* No. of comparisons to find the largest element of an unsorted array :

⇒ eg 6 5 10 7 9 18 2

let, max = 6 10 18 (n-1) comparisons → O(n)

* No. of comparisons to find the smallest element of an unsorted array.

⇒ eg 6 5 10 7 9 18 2

let, min = 6 5 2 (n-1) comparisons ⇒ O(n)

* Min. no. of comparisons to find largest & smallest element of an unsorted array :

⇒ eg 6 5 10 7 9 18 2

max = 6 10 18 (n-1) comparisons

min = 6 5 2 (n-2) comparisons

⇒ (n-1) + (n-2) = (2n-3) comparisons.

But, it's not minimum.

Date: _____ Page No: _____

• Another method (with less comparisons)

eg
10, 6 } max = ~~10~~ ~~12~~ 18 comparisons ⇒ ① (10-6)
min = ~~6~~ 3 ② (12-3)
12, 3 } 3 comparisons ③ (12-10)
④ (3-6)
9, 7 } 3 comparisons ⑤ (9-7)
⑥ (9-12)
18, 5 } 3 comparisons ⑦ (7-3)
⑧ (18-5)
⑨ (18-12)
⑩ (5-3)

$\Rightarrow 3\left(\frac{n}{2} - 1\right) + 1$

$\Rightarrow \frac{3n}{2} - 2$ comparisons if n = even.

eg
10
6, 12 } 3 comp. max = ~~10~~ ~~12~~ 18 ① ~~10~~ (12-6)
min = ~~10~~ ~~6~~ 3 ② (12-10)
③ (6-10)
3, 9 } 3 comp. ④ (3-9)
⑤ (9-12)
⑥ (3-6)
7, 18 } 3 comp ⑦ (7-18)
⑧ (18-12)
⑨ (7-3)

$\Rightarrow 3\left(\frac{n-1}{2}\right) \Rightarrow \frac{3n-3}{2}$ comp.

if n = odd.

→ –

* Complexity to find largest or smallest element of sorted array ⇒ O(1)

* If all the elements of array are distinct, complexity to find an element in unsorted array which is neither smallest nor largest.

⇒ Just compare any 3 elements.

eg [10 6 12] 3 9 7 18 21 3 1 7 99.

min = 6 ∴ 10 is neither smallest
max = 12 nor largest.

* thus, complexity will be O(1).

HW①
* Complexity of best-efficient algo to find the subarray with largest sum, (sub-array is contagious).

HW②
* Complexity of best-efficient algo to find the repeated elements of an unsorted array.

HW③
*

Date: Page No:

Binary Search: array must be sorted.

8	15	20	22	29	32	35	49
0	1	2	3	4	5	6	7

⇒ start = 0, end = 7, mid = $\frac{0+7}{2}$ = 3

① (49 - 22) ⇒ 49 > 22 ⇒ start = 4, end = 7, mid = $\frac{4+7}{2}$ = 5

② (49 - 32) ⇒ 49 > 32 ⇒ s = 6, e = 7, m = $\frac{6+7}{2}$ = 6

③ (49 - 35) ⇒ 49 > 35 ⇒ s = 7, e = 7, m = $\frac{7+7}{2}$ = 7

④ (49 - 49) ⇒ 49 = 49 (found)

Complexity of Binary Search:

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

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

After dividing the list, we apply binary search only in any one of the sublists.

Worst case ⇒ $O(\log n)$, Best case ⇒ $O(1)$

⇒
```
int top, bottom;
while ( top <= bottom)
{
    mid = (top + bottom) / 2;
    if (x == a[mid])
     printf ("found"); break;
     else if (x < a[mid])
      bottom = mid - 1;
     else
      top = mid + 1;
}
  if (top > bottom)
   printf ("not found");
```

Date: Page No.

Inversions in array :

if $(a[i] > a[j])$ if $i < j$

* it shows that the larger element is before the smaller one.

eg 10 6 12 15 9 1 ; how many inversions?

* 10-6, 10-9, 10-1, 6-1, 12-9, 12-1, 15-9, 15-1, 9-1

Ans ⇒ 9 inversions.

* Sorting techniques ultimately decrease the no. of inversions.

* min. no. of inversions in an array = 0

* max. no. of inversions in an array = $\frac{n(n-1)}{2}$

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

* The sorting technique which is highly affected by no. of inversions is insertion sort.

Date: Page No:

Que If there are n inversions in an array of size n what is the complexity of insertion sort?

Θ (n + no. of inversions)

DATA STRUCTURES

Date: Page No:

① Stack (LIFO)

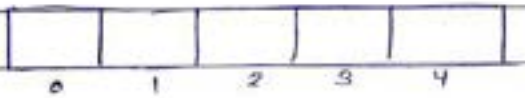
* Abstract Data Types (ADT) : They are just concepts (eg stack, queue, linked lists, tree, graph, etc.) but how to implement these, is not given.

* In C, there are 2 ways to implement any data structure.

① Static (Array) ② Dynamic (Linked Lists)

Static Implementation of Stack :

- TOP = −1 // empty stack



* For push, top++ then push
* For pop, pop then top--

Date: _____ Page No: _____

⇒ int a[max_size], top = -1;

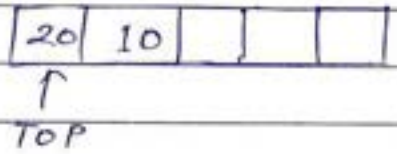
```
int isempty()                    // O(1)
{
    if (top == -1)
        return 1;    // true
    else
        return 0;    // false
}
int isfull()                     // O(1)
{
    if (top == max_size - 1)
        return 1;    // true
    else
        return 0;    // false
}
void push(int x)                 // O(1)
{
    if (isfull())
    printf("Overflow");
    else
    {
        top++;
        a[top] = x;
    }
}
int pop()                        // O(1)
{
    if (isempty()) printf("Underflow");
    else
    { x = a[top];
      top--; return x;
    }
}
```

Date: _____ Page No: ___

Que We are implementing the stack with the help of array such that TOP is always at index 0., then find the complexity of push & pop operations.

⇒

* Now, 20 is to be pushed., 10 has to be shifted. right



* Now, 20 is to be popped, then after popping 20, 10 has to be shifted towards left.

⇒ If TOP pointer is restricted to move then the elements have to be shifted for push & pop.

Thus, complexity of push ⇒ O (n)
Also, complexity of pop ⇒ O (n)

—x—

Date: Page No:

Applications of Stack :

① Expression Conversion & Evaluation
② Recursion

Expression Conversion :

a + b ⇒ infix for human
+ab ⇒ prefix { for
ab+ ⇒ postfix { systems

eg a + b * c / d

· priority ⇒ ① *, /, %.
② +, -

· All are left associative in C.

* In infix, brackets, priority, associativity exist but not in prefix & postfix.

Que a + b * c / d

Prefix ⇒ a + * bc / d
⇒ a + / * bc d
⇒ + a / * bc d

Postfix ⇒ a + bc * / d
⇒ a + bc * d /
⇒ a bc * d / +

Que assume *, / to be right associative.

Postfix ⇒ a + b * cd /
⇒ a + bcd / *
⇒ abcd / * + Ans

Date: Page No:

* Postfix Conversion (using stack):

- An array & a stack is required.
- In stack, we push the operators by using some rules;
  1. when stack is empty.
  2. When top of stack contains left parenthesis.
  3. When top of stack contains lower priority operator.
  4. When top of stack contains equal priority operator with right associativity.
  5. We can always push left parenthesis.

eg $a + b * c / d$

stack | + | ~~*~~ (pop) | / |

array | a b c * d / + |

Que $a + b * c - d \wedge e \wedge f$

priority
1. $\wedge$ ⇒ right assoc.
2. *
3. +
4. −

(2), (3), (4) ⇒ left ass.

stack | ~~+~~ | ~~*~~ | - | $\wedge$ | $\wedge$ |

array | a b c * + d e f $\wedge$ $\wedge$ - |

postfix algorithm is standard, but prefix algorithms are many in no.

Date:_____ Page No:____

Que $a + (b - c * d) / k$

stack | + | (| - | * | / |

array | a b c d * - k / + |

* on right parenthesis, pop until left parenthesis is popped.

* Prefix Conversion (using stack):

* Two stacks are used;
  1. Operator Stack - (rules are same as those of postfix conversion)
  2. Operand Stack

eg $a + b * c / d$

Operator | + | * | / |

Operand | a | b | c | * bc | d | / * bcd |

* bc, d

/ * bc d

* Before popping one operator, pop 2 operands from operand stack.

⇒ + a / * bcd Ans

Date: _____ Page No: _____

Que $a + b * c - d \wedge e \wedge f$

priority
① $\wedge$ ⇒ right ass.
② $*$
③ $+$ } left ass.
④ $-$

Operator | + | * | - | ^ | ^ |

Operand | a | b | c | *bc | +a*bc | d | e | f | ^ef | ^d^ef |

Ans ⇒ $- + a * b c \wedge d \wedge e f$

Expression Evaluation:

* Postfix Expression Evaluation:

eg $2\ 4\ 3\ -\ +$

| 2 | 4 | 3 | 1 | 3 |

* Read from LHS & push(x) on stack operands.
* On operator, pop 2 operand

Ans ⇒ 3

$2 + 1 = 3$ ← $\quad 4 - 3 = 1$ ←

Que $8\ 2\ 3\ \wedge\ /\ 2\ 3\ *\ +\ 5\ 1\ *\ -$

⇒ | 8 | 2 | 3 | 8 | 1 | 2 | 3 | 6 | 7 | 5 | 1 | 5 | 2 |

Ans ⇒ 2

$2 \wedge 3 = 8$ ←, $8 / 8 = 1$ ←, $2 * 3 = 6$ ←, $1 + 6 = 7$ ←, $5 * 1 = 5$ ←, $7 - 5 = 2$ ←

Date: Page No:

* Prefix Expression Evaluation:

eg + - 3 2 6 * Read from RHS

| 6 | 2 | 3 | 1 | 7 |

Ans ⇒ 7

3 - 2 = 1 , 1 + 6 = 7

⇒ Space Complexity

* for (i=1; i<=n; i++) ⇒ O(1) (only 2 variables)
  printf ("%d", i);

* for (i=1; i<=n; i++) ⇒ O(n)
  printf ("%d", a[i]);

* Generally, iterative programs, if they don't use any dynamic data structure (array, linked list, tree, graph, etc.), have space complexity O(1).

⇒ To convert prefix to postfix or viceversa, first we'll convert to infix.

• Prefix → from L.H.S, postfix → from R.H.S
⇒ solve an operator if it's followed by 2 operands.

eg + a / * b c d
+ a / (b * c) d
+ a ((b * c) / d)
a + ((b * c) / d)
⇒ a + b * c / d
(infix)

eg * + b c - d e
⇒ ((b+c) * (d-e))
⇒ (b+c) * (d-e)

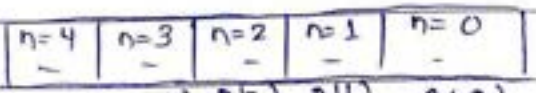
* Here, all brackets can't be dropped

Date:_____ Page No:___

* How to find space complexity of recursive programs

eg
```
void P(int n)
{
    if (n<=0)
     return ;
    else
    { P(n-1);
      printf ("%d", n);
    }
}
```

space complexity is equal to size of stack.



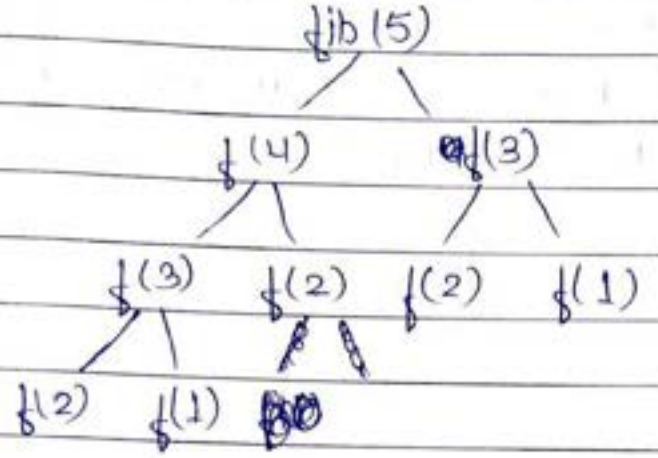
⇒ 5 for P(4)

⇒ (n+1) for P(n)
i.e. O(n)

Que
```
int fib (int n)
{
    if (n==1 || n==2)
     return 1 ;
    else
    return fib(n-1) + fib(n-2) + 1;
}
```

* For space complexity, make a recursion tree, the depth of longest branch is the space complexity.

Date:_____ Page No:_____

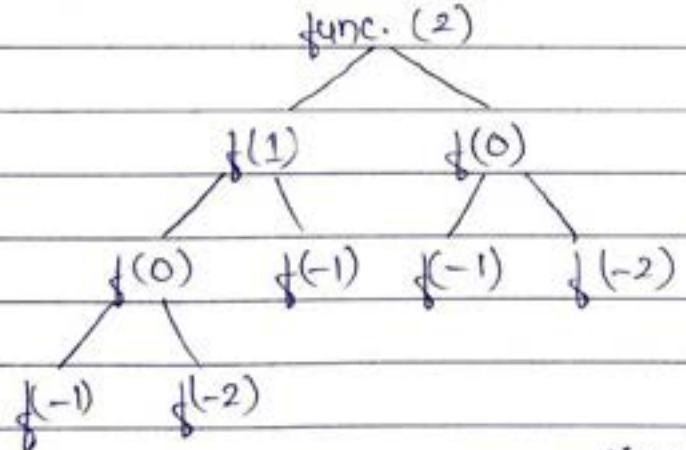


longest branch
⇒ 4 for fib(5)
ie.
⇒ n-1 for fib(n)
ie. O(n)

* Sometimes, we cannot write recurrence relation for a recursive code, then how to find time complexity.

⇒ Make recursion tree, & try to find some relationship between parameter of function and no. of nodes in tree.

eg



- n = 2
- no. of nodes = 8

∴ relation can be,
⇒ $2^{2+1} = 2^3 = 8$

time. comp. ⇒ 2^{n+1} ⇒ $O(2^n)$

* Infinite recursion: Because of limitation of stack size (main memory), recursion can't be practically infinite.

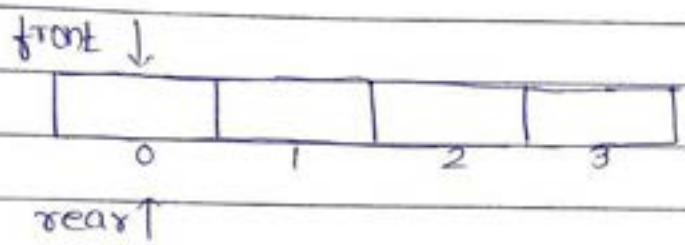
Date: _____ Page No: ____

(2) Queue (FIFO)

* We maintain 2 pointers - ① rear (for insertion)
② front (for deletion)

Static Implementation of Queue:

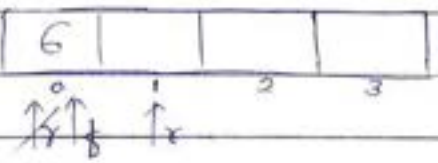
* initially, front = rear = 0 (empty queue)



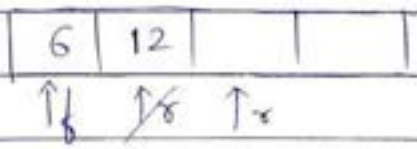
* For enqueue ⇒ a[rear] = x & rear++

* For dequeue ⇒ x = a[front] & front++

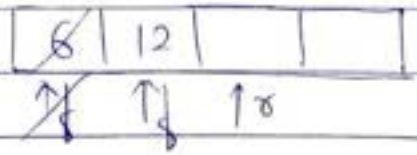
eg Insert 6



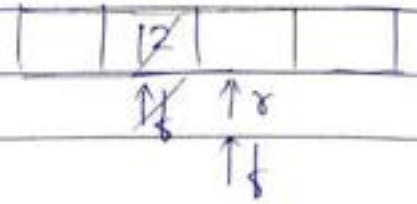
Insert 12



delete 6

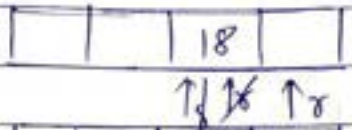


delete 12

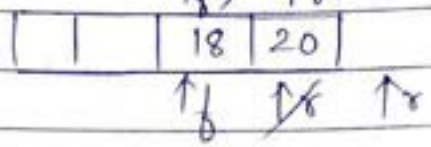


front = rear means queue is empty

Now, insert 18

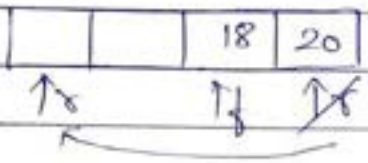


Insert 20



* Now, rear pointer is out of the queue, thus, although queue is not full, insertion can't be done.

⇒ Thus, circular queue is used.



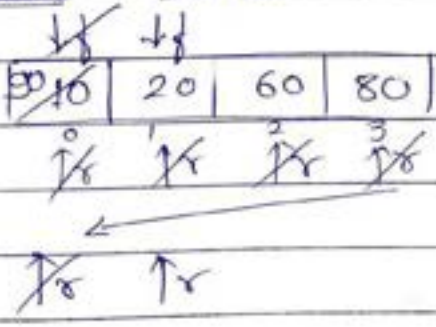
⇒ int a[max_size], rear = front = 0;

```
int isempty()          // O(1)
{
    if (rear == front)
        return 1;      // true
    else
        return 0;      // false
}
int isfull()           // O(1)
{
    if (rear == max_size - 1)
        return 1;      // true
    else
        return 0;      // false
}
```

Date: _____ Page No. _____

```
void enqueue (int x)          // O(1)
{   if (isfull())
    printf("Overflow");
    else
    {  a[rear] = x;
       rear++;
    }
}
void dequeue ()               // O(1)
{   int x;
    if (isempty())
    printf("Empty");
    else
    {  x = a[front];
       front++; return x;
    }
}
```

Circular Queue :

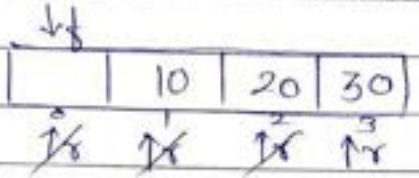


⇒ insert 10, 20, 60
delete 10
insert 80
insert 90

Now, front = rear & queue is full & the same condition represents that queue is empty. Thus, we'll go for another implementation.

* If size of array is n, then the queue can have max. (n-1) elements as one index(n) is always empty. ~~pointed by 'front'~~

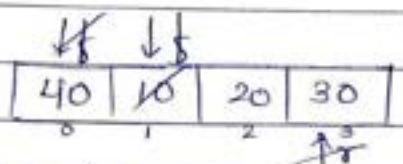
⇒ One index of array is always empty. & initially that index = 0. Also, front = rear = 0.



→ Insert 10 ⇒ inc. rear then insert

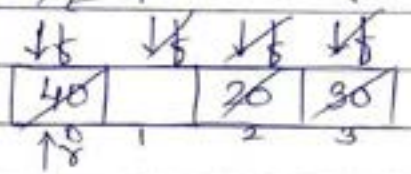
Insert 20, 30

Now, queue is full.



→ Delete 10. ⇒ inc. front & then del

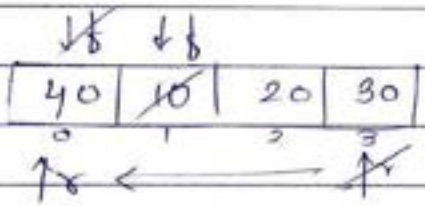
Insert 40 ⇒ rear back to index 0



→ Delete 20, 30

Delete 40 ⇒ front back to index 0

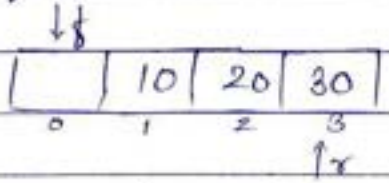
Now, front = rear means queue is empty



⇒ Insert 10, 20, 30
Delete 10
Insert 40
Now queue is full

- Condition of Queue Overflow ⇒ [rear + 1 = front]

But,



Here, queue is full
but, rear + 1 ≠ front (3+1 ≠ 0)

Thus, (rear+1) mod 4 = front ((3+1) mod 4 = 0)
(array size)

Date: _____ Page No: _____

```
int a[N];          // N is array size
int rear = front = 0;

void enqueue (int x)          // O(1)
{
  if ((rear+1) % N == front)
    printf ("Overflow");
  else
  {  rear = (rear + 1) % N;
     a[rear] = x;
  }
}

void dequeue ()               // O(1)
{
  if ( rear == front)
  printf ("Empty");
  else
  {  front = (front + 1) % N;
     x = a[front];
  }
}
```

Date: Page No:

Que Can we implement queue using stack?

⇒ Yes, 2 stacks are required (say S1 & S2)

S1 | 10 20 30

S2 | 30 20 10

Insert 10, 20, 30
Delete 10

* For enqueue, push element in S1.
* For dequeue, pop element from S2.
* If S2 is empty, pop all elements from S1 & push them one by one in S2.

* Complexity → Enqueue → O(1)
  Dequeue → O(n); all elements have to be shifted from S1 to S2
  [Best case : O(1)]

Que Can we implement stack using queue?

⇒ Yes, 2 queues are required (say Q1 & Q2)

Q1 | 10 20 10

Q2 | 10

Insert 10
Insert 20
Delete 20

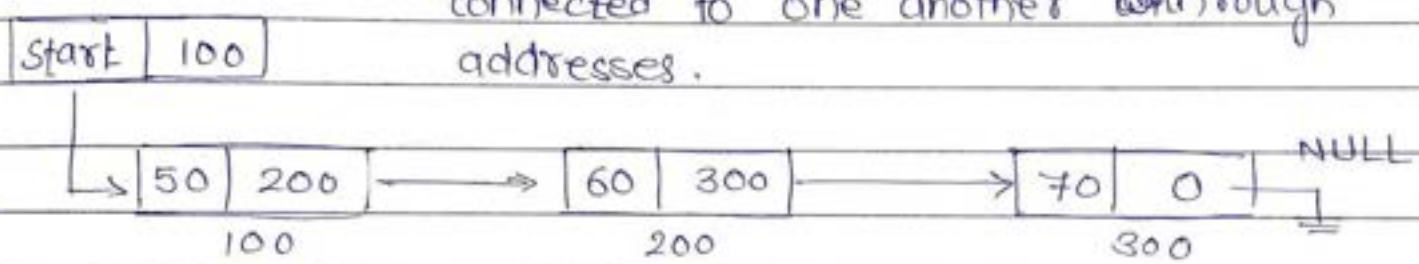
* for push, insert element in Q1, but before that, delete all the elements from Q1 & insert them in Q2.
* Now, after insertion, shift all elements back from Q2 to Q1
* for pop, simply delete element from Q1

start ptr. = Head / top

Date: Page No:

* complexity ⇒ Push ⇒ O(n) ⇒ O(n+n)
Pop ⇒ O(1)
Q1 to Q2, Q2 to Q1

③ Linked List : A data structure in which nodes are connected to one another through addresses.



* Linked list nodes are non-contagious while array is stored contiguously.

• In array, $(i+1)^{th}$ element = base address + i (index) * w (size of data type)

eg 

4^{th} element = 0 + 3 * 1 = 3

* In array, elements can be accessed randomly or directly. while in linked lists elements are sequentially accessed.

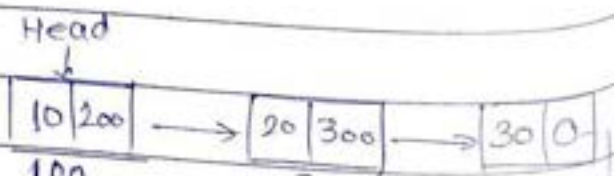
* complexity to read last element of array ⇒ O(1)
& of linked list ⇒ O(n)

Date: Page No:

⇒
```
struct node
{
    int data;
    struct node *next;
};

struct node *head = NULL;

void insert (int x) (in beginning)
{
    struct node *p1;
    p1 = (struct node *) malloc (sizeof (struct node));
    p1 -> data = x;
    p1 -> next = head;
    head = p1;
}

void traverse ()          // O(n)
{
    struct node *p1;
    p1 = head
    while (p1 -> next != NULL)
    {
        printf ("%d", p1 -> data);
        p1 = p1 -> next;
    }
}
```

// those structures which contains reference of itself known as self referential structures. They're generally used to implement tree, graph, linked list, etc.

But, it'll not print the last node.



→ 10 20

thus, while (p1 != NULL)

→ 10 20 30

Date: _____ Page No: _____

```
void search (int x)          // O (n)
{
   struct node *P1;
    P1 = head;
   while( P1 -> data != x  && P1 != NULL)
        P1 = P1 -> next;
   if (P1 == NULL)
    printf(" Not found");
    else
     printf ("found");
{
```

```
void delete ()  (in beginning)
{
   struct node *P1 = head;
    head = head -> next;
     free (p1);
{
```

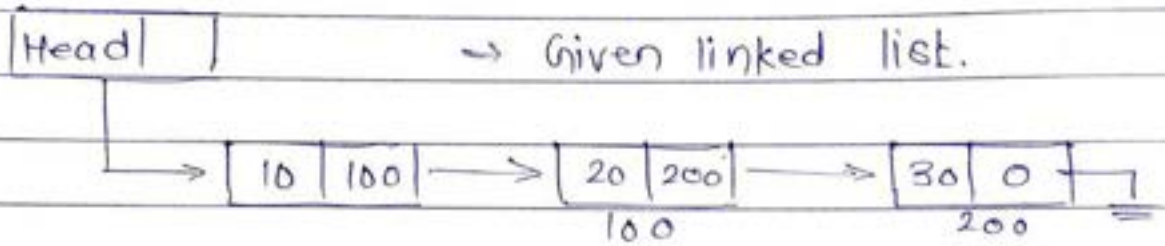
* In array, binary search can be done if it's sorted but in linked list if it's sorted, binary search can't be applied as we cannot find mid because it's not contagious. Thus, mid has to be traversed i.e. O(n/2) or O(n)

* If an operation requires traversal of linked list, complexity will be O(n) else O(1).

Date: _____ Page No: _____

Que
```
void doSomething (struct node *Pl)
{
  if (Pl != NULL)
  {
    doSomething (Pl -> next);
    printf ("%d", Pl -> data);
  }
}
```

Find O/P if doSome (Head) is called.

⇒ 

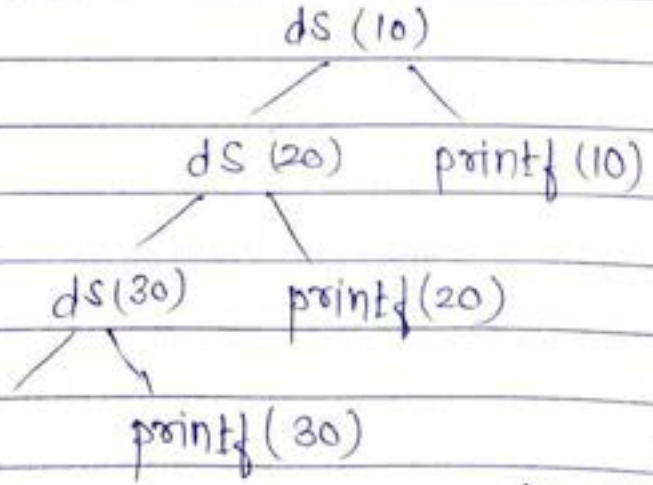
→ Given linked list.

⇒

Pl → 10	Pl → 20	Pl → 30	Pl → 0
printf	printf	printf	
ds(10)	ds(20)	ds(30)	0

⇒ 30 20 10

OR

→ 

⇒ 30 20 10

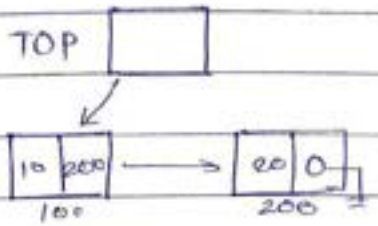
→ -

* Dynamically created memory is not destroyed after function call.
end of

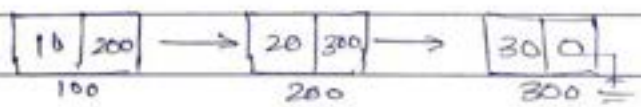
Date: Page No:

Stack using Linked List:

```
struct node
{ int data;
  struct node *next;
};
struct node *top = NULL;

void push (int x)        // O(1) (beginning)
{
struct node *P1;
P1 = (struct node*) malloc
     (sizeof (struct node));
P1 -> data = x;
P1 -> next = top;
top = P1;
}
void pop ()              // O(1) (beginning)
{ struct node *P1;
  if (top == NULL)
  printf ("Underflow");
  else
  { P1 = top;
    top = top -> next;
    printf ("%d", P1 -> data);
    free (P1);           // to avoid
  }                      memory leakage
}
```



push 30 → O(n)
(push in end)

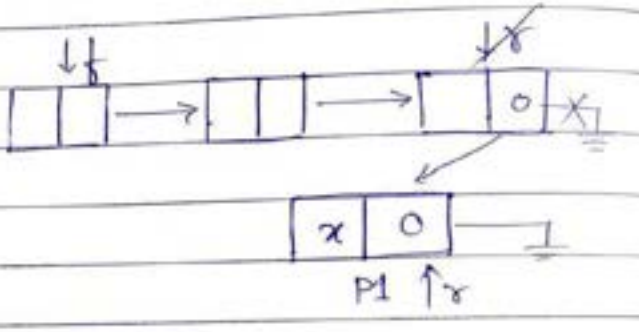


pop 30 → O(n)
(pop in end)

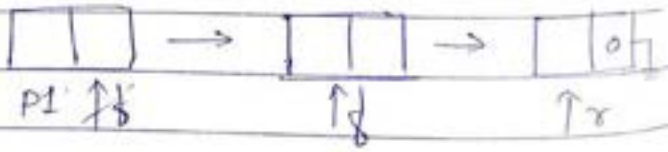
* Thus, we'll do in the beginning.
* malloc creates a memory location & returns its address

Queue using Linked List:

```
void enqueue (int x)        // O(1)
{
  struct node *P1 = (struct node *)
   malloc(sizeof (struct node));
   P1 -> data = x;
   P1 -> next = NULL;
   if (rear == NULL)
     front = rear = P1;
     else
   {   rear -> next = P1;
       rear = P1;
   }
}
```




```
void dequeue ()        // O(1)
{
   struct node *P1;
if(front != NULL) P1 = front;
   front = front -> next;
     free (P1);
   else
   { printf("Underflow"); }
}
```

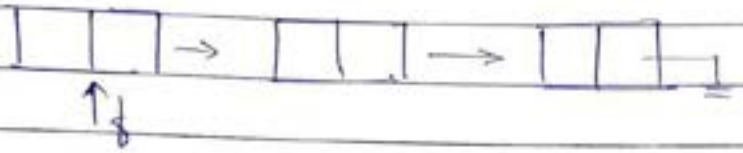


* With both front & rear, enqueue = O(1)
dequeue = O(1)

Date: Page No:

Que We are implementing queue using singly linked list, what will be the complexity of enqueue & dequeue if:

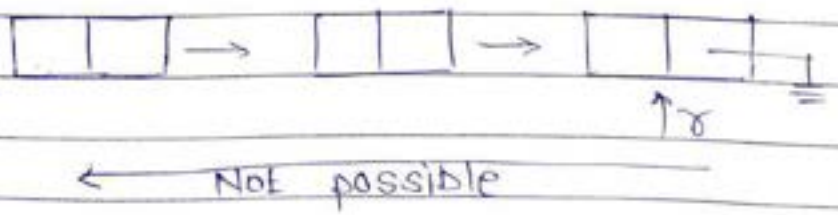
(a) We are maintaining only one pointer i.e. front, which points to the first node of linked list.

⇒ 

- Dequeue ⇒ O(1) ⇒ just delete & increment front
- Enqueue ⇒ O(n) ⇒ traverse entire list upto the end to insert a new node.

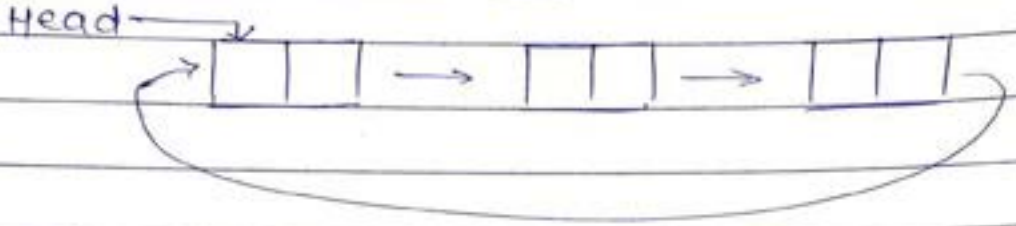
(b) We are ~~implementing~~ maintaining only rear pointer, which points to the last node of linked list.

⇒ - Enqueue ⇒ O(1) ⇒ just add & increment rear.
- Dequeue ⇒ Not Possible ⇒ For this, we've to access first node & in singly linked list, we can't traverse from last node to first node



* we can't access first node as we can't have its address.

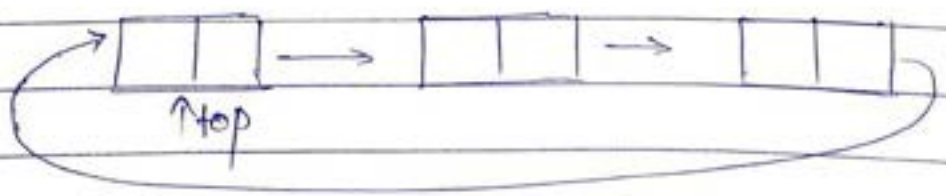
Circular Linked List :




```
void traverse ()
{   struct node *p1;
    p1 = head;
    do
    {  printf("%d", p1->data);
       p1 = p1->next ;
    }
    while (p1 != head);
}
```

Que What is the complexity of push & pop if the stack is implemented by circular linked list & top points to the first node of linked list.

⇒ Push ⇒ O(n) , Pop ⇒ O(n)



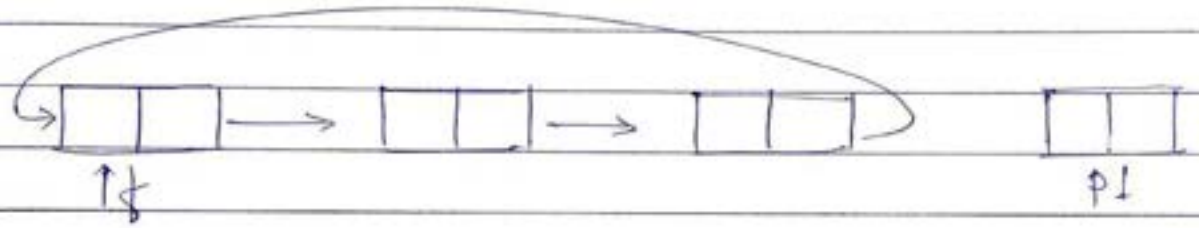
* For push & pop, we've to traverse entire list upto the last node, to change the 'next' of last node.

Date: Page No:

Que We are implementing queue using circular linked list, what will be the complexity of enqueue & dequeue if;

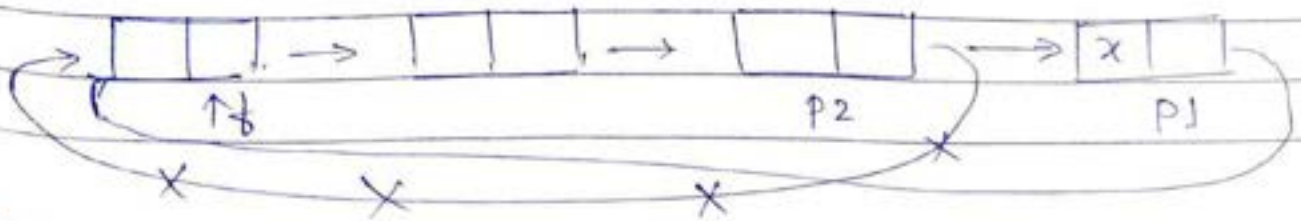
(a) We are maintaining only one pointer ie. front, which points to the first node of linked list. & enqueue is performed at the end of linked list. & dequeue from the beginning.

⇒



- Enqueue ⇒ O(n) ⇒ traverse upto last node & then insert.
- Dequeue ⇒ O(n) ⇒ after deleting first node, we've to change "next" of last node

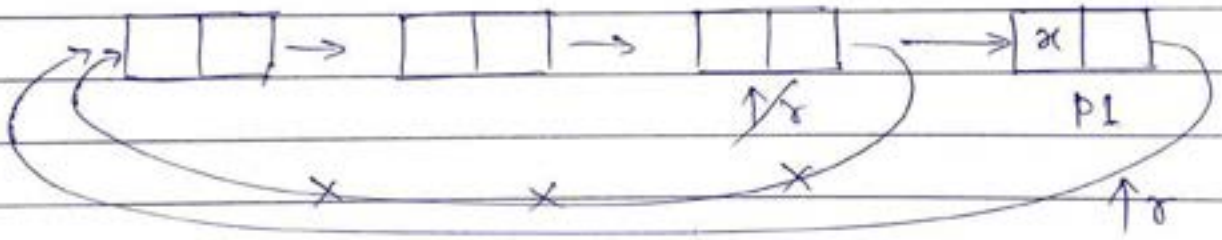
```
Enqueue ⇒ struct node *p1 = (struct node *)
          malloc (sizeof (struct node));
          p1 -> data = x;
          p2 = front;
          while (p2 -> next != front)
            p2 = p2 -> next;
          p2 -> next = p1;
          p1 -> next = front;
```



(b) We are maintaining only rear pointer which points to the last node of linked list. & enqueue is performed at the end & dequeue from the beginning of the linked list.

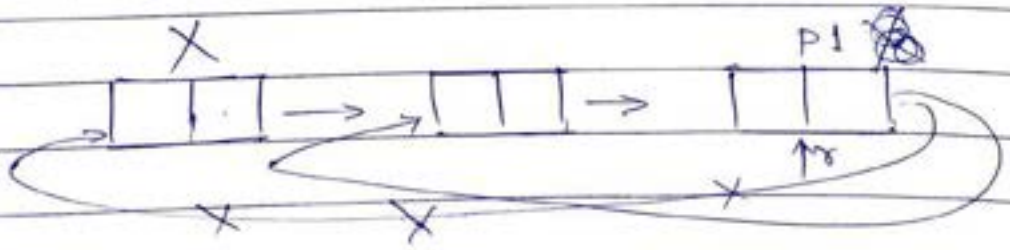
⇒ • Enqueue ⇒ O(1) ⇒

```
struct node *P1 = (struct node*)
malloc (sizeof(struct node));
  P1→ data = x ;
  P1→ next = rear→ next;
   rear→ next = P1 ;
    rear = P1;
```



• Dequeue ⇒ O(1)

```
*P1 = rear
 rear→ next = rear→ next→ next ;
```

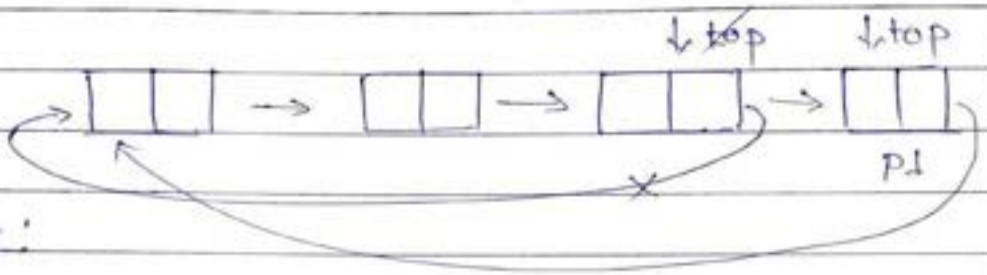


* when top is at last node but push & pop is done at the beginning. ⇒ Push ⇒ O(1), Pop ⇒ O(1)

Date: Page No:

Que We are implementing stack using circular linked list. What will be the complexity of push & pop & top points to the last node of linked list.

⇒ • Push ⇒ O(1)




```
p1 -> next = top -> next;
top -> next = p1;
top = p1;
```

• Pop ⇒ O(n)


```
p1 =
```

Date: Page No:

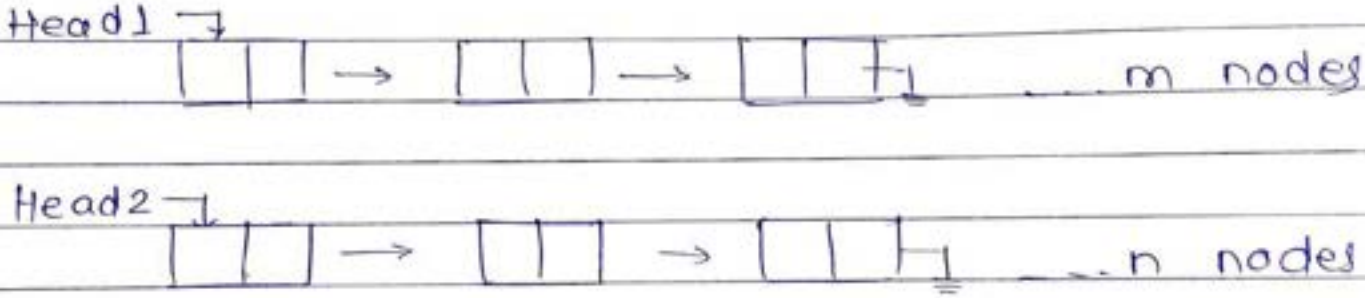
Set → repeatation is not allowed

Que We are implementing set using linked list, what will be the complexity of:

(a) Membership ⇒ searching entire list ⇒ O(n)

(b) Cardinality ⇒ Traverse entire list ⇒ O(n)

(c) Union ⇒



- traverse all nodes of list 1 i.e. m elements
  Now, to avoid repeatation, for ~~e~~ each element of list 2, check if it is in list 1.

  Thus ⇒ O(m * n) or O(n²)

(d) Intersection ⇒ O(m * n) or O(n²)

(e) Insertion ⇒ O(n) ⇒ After insertion, entire list has to be traversed to avoid repeatation.

Date: _____ Page No: _____

* If we use extra data structure (let's use hash table) then complexity of union will be, (& intersection),

⇒ traverse the list 1 & insert all elements in hash table.
Now, traverse list 2 & delete the elements from hash table if it is in list 2.
Now, add elements of hash table in list 2 for union

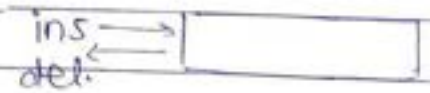
eg • 2 3 4 • Hash table
• 5 4 6 ⇒ 2, 3, ~~4~~ • searching in hash table → O(1)

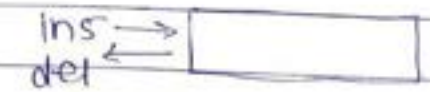
list 2 ⇒ 5, 4, 6, 2, 3

⇒ O(n)

—x—

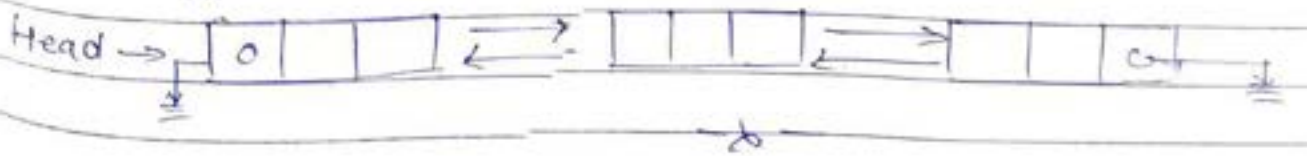
Double Ended Queue

① Input Restricted Queue 

② Output Restricted Queue 

—x—

Doubly Linked List



—x—

Date: Page No:

Priority Queue :

- Insertion ⇒ no restriction ; anywhere.
- Deletion ⇒ Ascending ⇒ Delete smallest element first
  Descending ⇒ Delete largest element first

* To implement priority queue, heaps are used.
  - For ascending priority queue, min heap is used
  - For descending priority queue, max heap is used.

Que Originally, priority queue is implemented by heap.
But, we are implementing it (ascending priority queue) by (a) unsorted array, (b) sorted array.
Find the complexity of insertion & deletion.

(a)
⇒ Unsorted Array ⇒ Insertion ⇒ O(1)

⇒ Deletion ⇒ $\underset{\downarrow}{n} + \underset{\searrow}{n}$ ⇒ O(n)

1 pass of selection sort — shifting all elements towards le

(b)
⇒ Sorted Array ⇒ Insertion ⇒ O(n)

⇒ Deletion ⇒ O(n) (if array is in ascending order)

O(1) (if array is in desc.

Date: Page No:

Que We are implementing priority queue with the help of heap. What will be the complexity of decrease key operation if

1a) index of key to be decreased is given

1b) index of key is not given.

(a) ⇒ After decrease key, reheapification will take $O(\log n)$.

(b) ⇒ $\underbrace{n}_{\text{searching}} + \log n \Rightarrow O(n)$ (↳ reheapification)

Ans We've to perform n search operations, $\sqrt{n}$ delete min operations, $\sqrt{n}$ insert operations, n decrease key operations & a pointer is provided for decrease key ie. no need to search that key. Which data structure will you choose?

(a) Unsorted Array ⇒ Search $(n*n)$, Delete $(\sqrt{n}*n)$, Insert $(\sqrt{n}*1)$, Decrease $(n*1)$, Total $\Rightarrow O(n^2)$

- to search min element + shifting $(n+n=n)$

(b) Sorted Array ⇒ $(n*\log n)$ (binary search), $(\sqrt{n}*n)$ shifting, $(\sqrt{n}*n)$ shifting (re-sort), $(n*n)$ re-sort $\Rightarrow O(n^2)$

But, If array is in descending order ⇒ $O(1)$ no need for shifting

Date: Page No:

For resort,

* In sorted array, insert at the end & then apply 1 pass of insertion sort.

(c) Heap ⇒ $(n * n)$ $(\sqrt{n} * \log n)$ $(\sqrt{n} * \log n)$ $(n * \log n) = O(n^2)$

reheapification

(d) Unsorted ⇒ $(n * n)$ $(\sqrt{n} * n)$ $(\sqrt{n} * n)$ $(n * 1)$ ⇒ $O(n^2)$

Linked list

to search min element

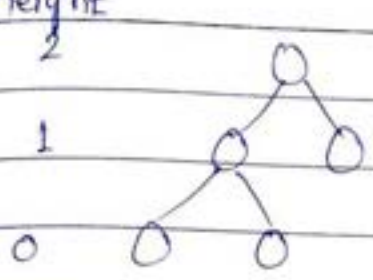
⇒ Any data structure can be used ← Ans

TREES

Date: _____ Page No: _____

* Stacks, queues, linked lists, etc. are linear data structures & trees & graphs are non-linear.

* A binary tree is a tree in which a node can have atmost 2 child nodes.



* Highest possible level is depth of tree ie. 2, here

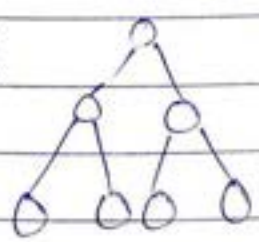
* Max. nodes at level d ⇒ 2^d

* Max. nodes in a tree of depth d ⇒ $2^{d+1} - 1$ $(2^0+2^1 \cdots 2^d)$

* Min. nodes in a tree of depth d ⇒ d+1

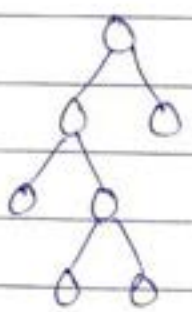
• Complete Binary Tree : All the leaf nodes should be at same level & every non leaf node should have exactly 2 child nodes.

OR ⇒ Every non leaf should have exactly 2 child nodes (strict binary tree)

① 

Complete

OR ⇒ ②



Complete

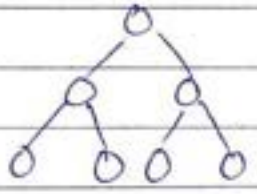
Also ⇒ strict (either 0 or 2 child)

* Strict Binary tree : Each node has 0 ...

Date _____ Page No. _____

* Leaf nodes are external nodes & non-leaf are internal nodes.

* If there are n internal nodes in a complete binary tree, there will be n+1 external nodes.



3 internal & 4 external

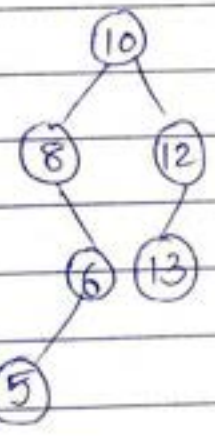
* In ternary complete tree,
  internal ⇒ n , external ⇒ 2n+1

* In quaternary complete tree,
  internal ⇒ n , external ⇒ 3n+1

* In k-ary complete tree,
  internal ⇒ n , external ⇒ (k-1)n + 1

Tree Traversal

eg



- Inorder (L Root R) ⇒ 8 5 6 10 13 12
- Preorder (Root L R) ⇒ 10 8 6 5 12 13
- Postorder (L R Root) ⇒ 8 5 6 8 13 12 1

In such questions, inorder will be given & preorder or postorder will be given.

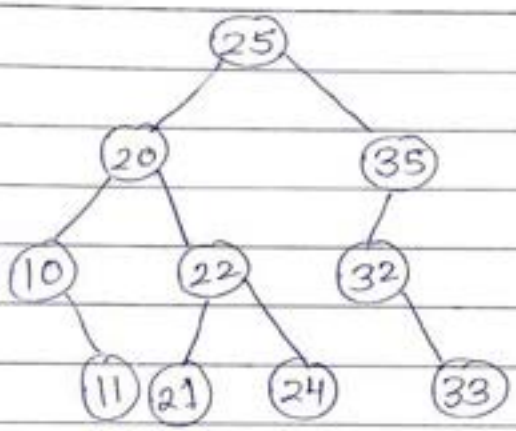
Date:_____ Page No:____

Que Inorder : 10 11 20 21 22 24 25 32 33 35

Preorder : 25 20 10 11 22 21 24 35 32 33

root

⇒



* Inorder tells about position of nodes.
* Pre/Post tells about which node comes first.

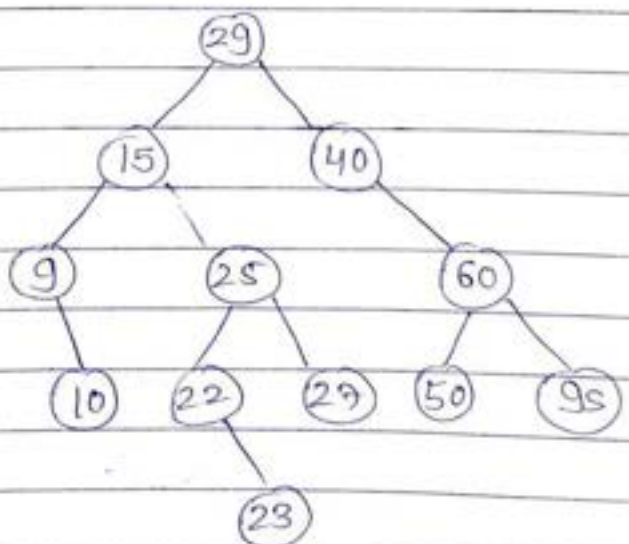
CS. 2007 Que 39

CS. 2013 Que 32

Que Inorder : 9 10 15 22 23 25 27 29 40 50 60 95

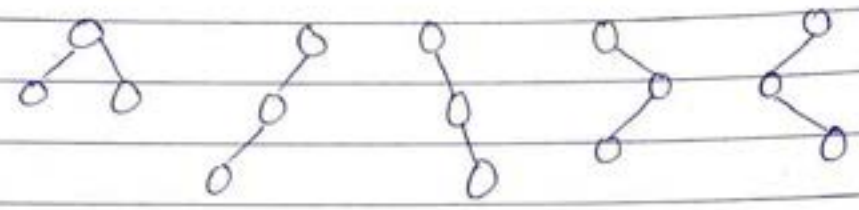
Postorder : 10 9 23 22 27 25 15 50 95 60 40 29

⇒



Date: _____ Page No: _____

* If you have 3 keys, how many different binary trees are possible?

⇒ 

Ans ⇒ 5

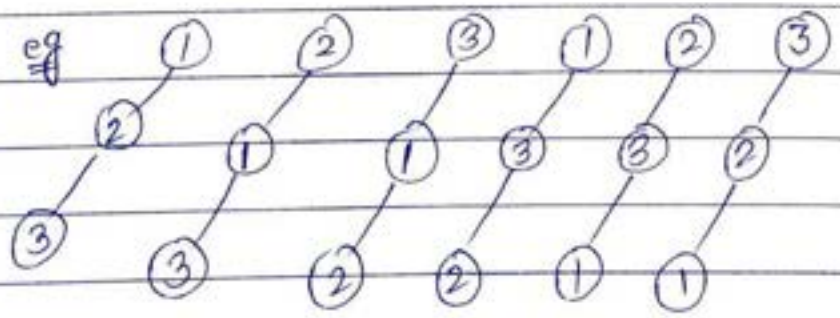
* If you have n keys, then no. of different binary trees is,

$$\Rightarrow \frac{^{2n}C_n}{n+1}$$

* If you have n distinct keys, then no. of different labelled binary trees is,

$$\Rightarrow \frac{^{2n}C_n}{n+1} \times n!$$

= ways to label a tree of n nodes

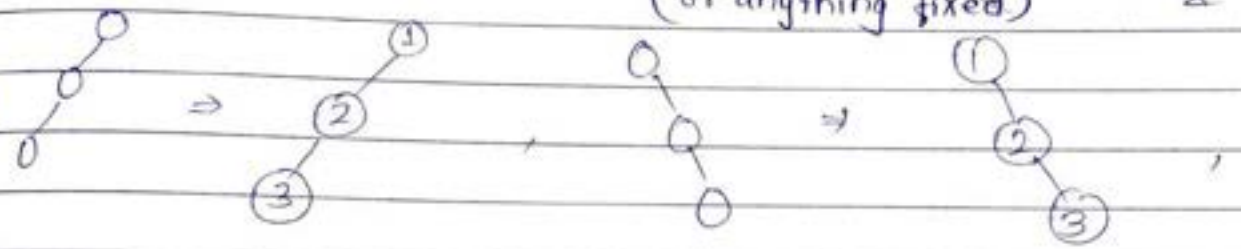
eg 

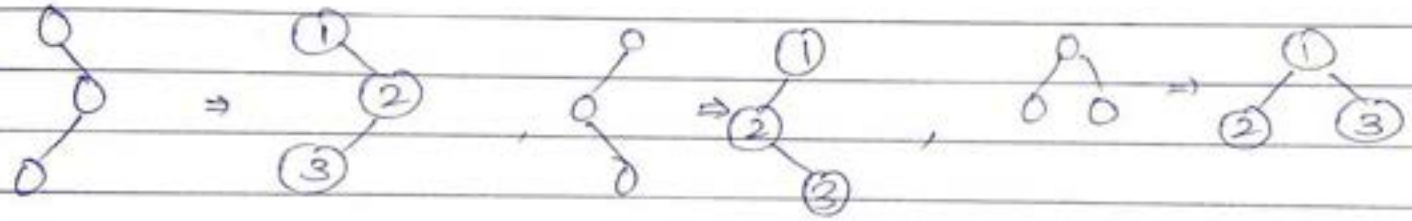
$3! = 6$

Date: _____ Page No: _____

* If you have 3 keys, you'll have $5 \times 3! = 30$ labelled binary trees out of which how many trees have their preorder output to be 1 2 3. (also for inorder & postorder)
(or anything fixed)

⇒



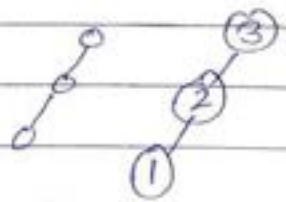


* ⇒ $\frac{2nC_n}{n+1}$; for n keys & preorder → 1 2 3 ... n

—x—

* If you have n keys, then how many ways to label one tree such that it becomes a BST.

⇒ only one eg



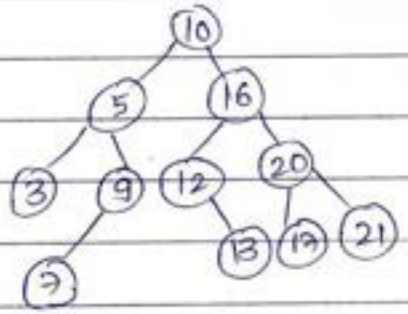
BST gives O(logn) searching & it's a [illegible] which isn't possible in linked list.

Date: Page No.

Binary Search Tree : It is a binary tree in which value of left child is less than that of parent & value of right child is greater or equal to parent.

eg 10 16 12 5 9 3 7 13 20 17 21

→



Inorder → 3 5 7 9 10 12 13 16 17 20 21

* Inorder traversal of BST is always sorted in order.

→ —

* If you have n distinct keys, how many different BST are possible?

→ $\frac{2n C_n}{n+1}$ (in BST, inorder is sorted (1, 2, 3 ... n))

CS-2011
Que 44
* [illegible] 29

* In BST, if one of the preorder and postorder is given & you are asked to find the other one then sort it to find inorder.

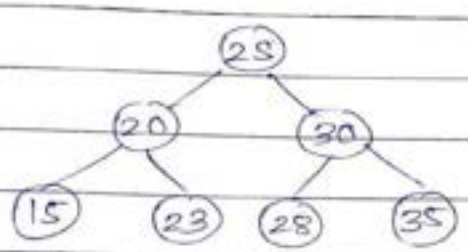
eg preorder : 5 4 9 6 0
inorder → 0 4 5 6 9

Now postorder can be found easily.

Date: _____ Page No: _____

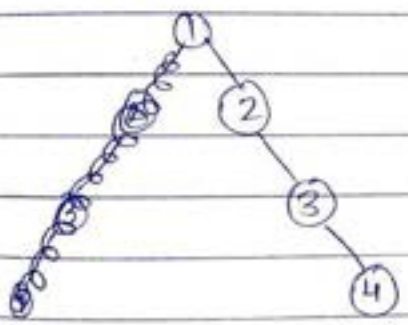
Complexity of BST :

Search



- Best case ⇒ O(1) (Root)
- Average case ⇒ O(logn)

* If it takes a form of chain ie input is sorted



- Worst case ⇒ O(n)

∴ Complexity ⇒ O(depth)

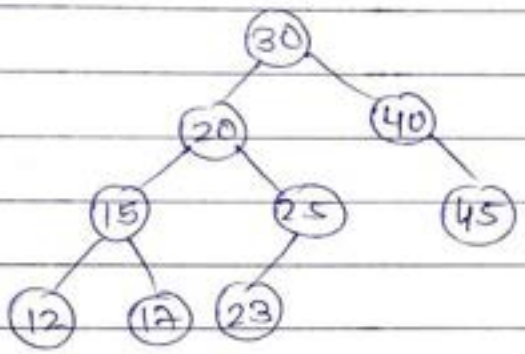
Insertion ⇒ For insertion, we've to traverse upto the depth of BST.

- Best case ⇒ O
- Average case ⇒ O(logn)
- Worst case ⇒ O(n) (chain)

∴ Complexity ⇒ O(depth)

* Deletion also has the same complexity.

* Deletion in BST:



Case ① : Deletion of leaf node: simply delete the leaf node O(1)

Case ② : Deletion of non-leaf node having only 1 child node.

⇒ Replace the deleted node with its child node

Case ③ : Deletion of non-leaf node having both child nodes.

⇒ Replace the deleted node with either inorder successor or inorder predecessor.

* Inorder predecessor doesn't have right child & inorder successor doesn't have left child.

Date: Page No:

Que Suppose, you are searching 60 in a BST, can following keys be encountered during searching?

⇒ 10 100 90 20 25 80 75 45 60

- $x > 10\ 20\ 25\ 45$
- $x < 100\ 90\ 80\ 75$

Ans → Yes

Another Method ⇒ 10 100 90 20 25 80 75 45 60

① smaller than 60 ⇒ 10 20 25 45 → it should be in ascending order

② larger than, 60 ⇒ 100 90 80 75 → it should be in descending order

⇒ Then, it will be a valid search path.

31.2008

Que 71, 72, 73

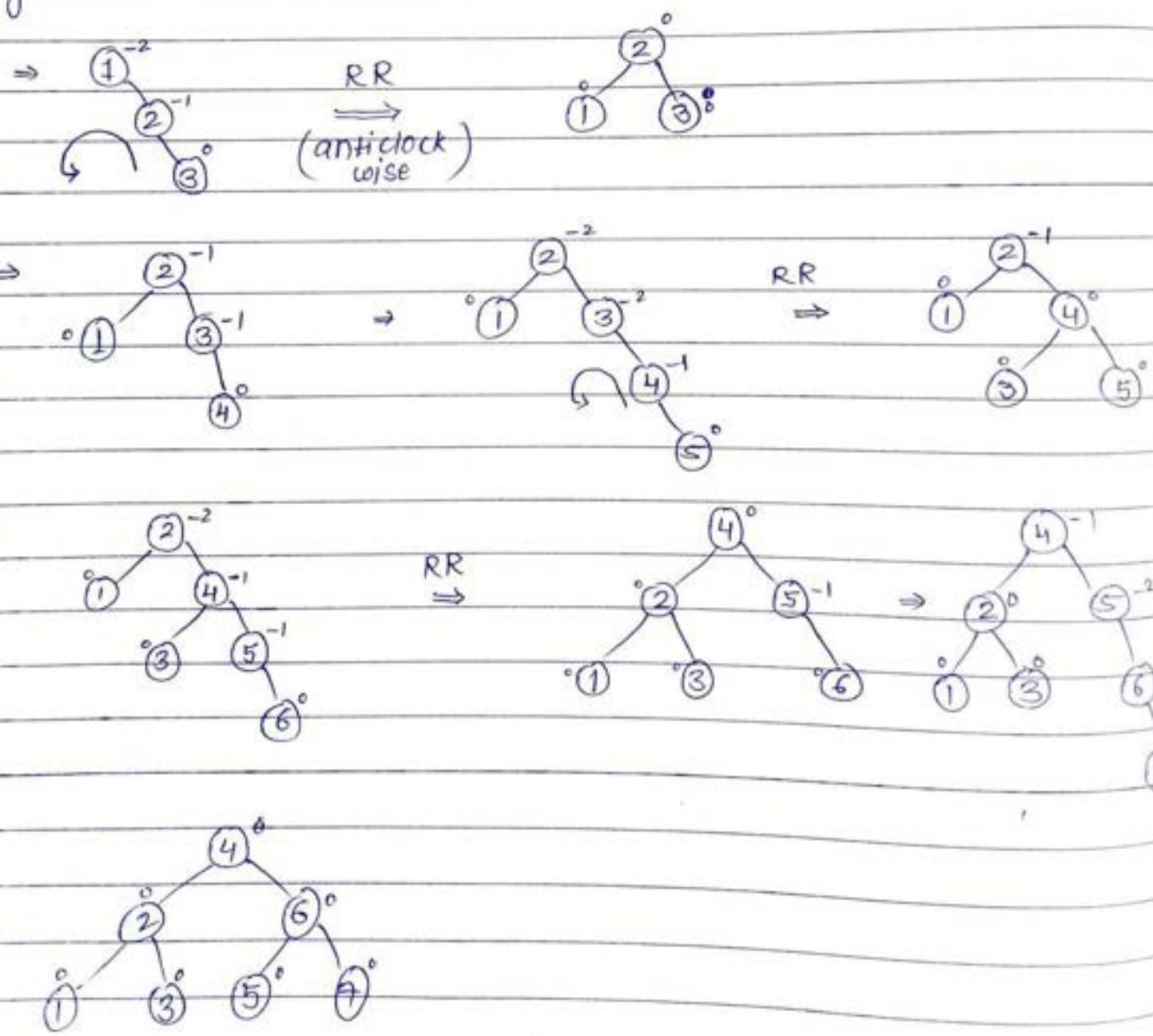
BST may take a form of chain but AVL tree can't. thus, they're introduced. Date: Page No.

AVL Tree (Height Balance Tree) : They are Balanced BST.

Balance = Height of Left Subtree − Height of Right Subtree

(0, -1, 1)

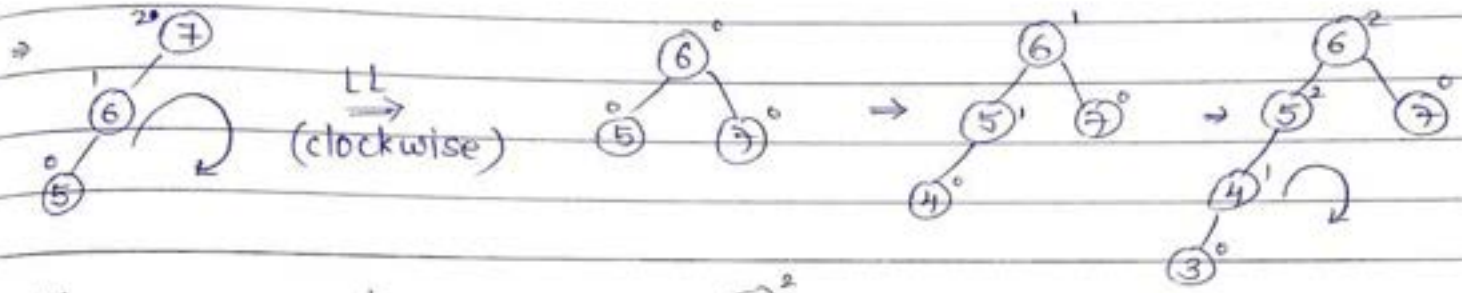
eg 1 2 3 4 5 6 7

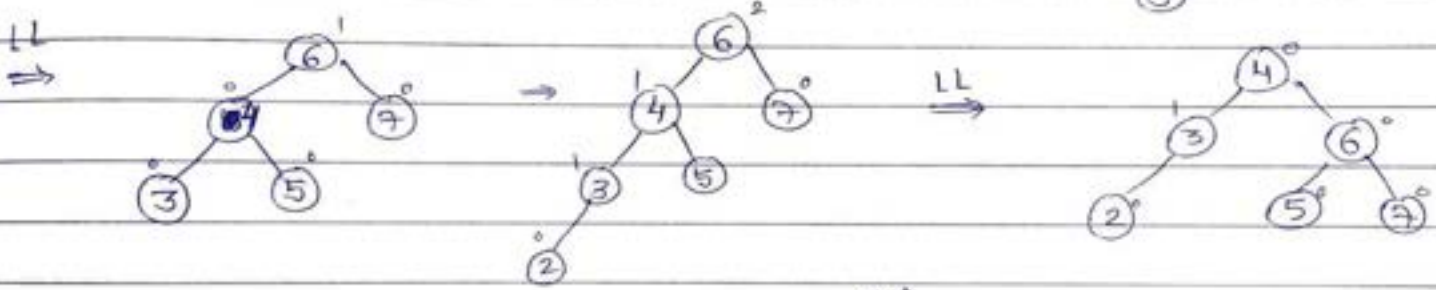


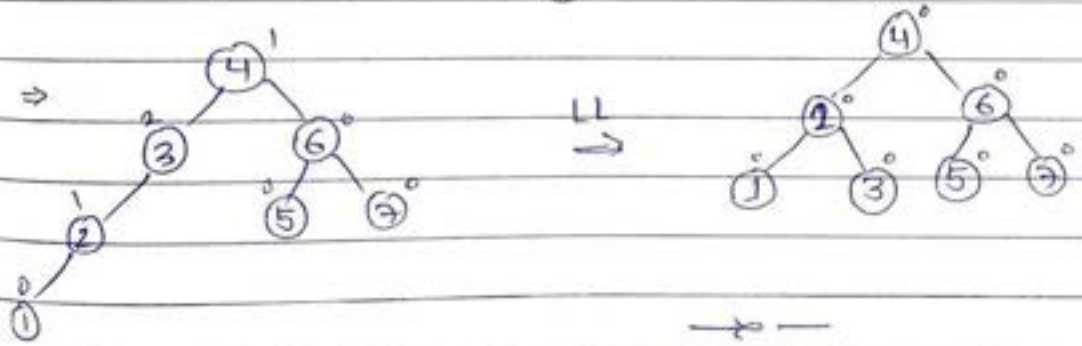
LR & RL ⇒ we have to rotate twice

Date _____ Page No: _____

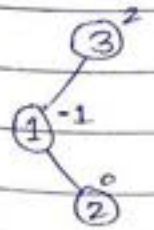
eg 7 6 5 4 3 2 1



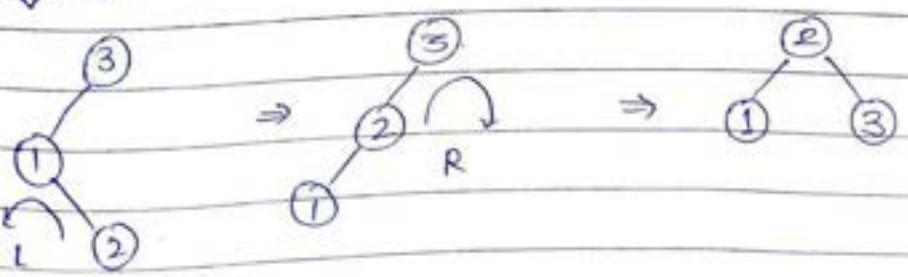


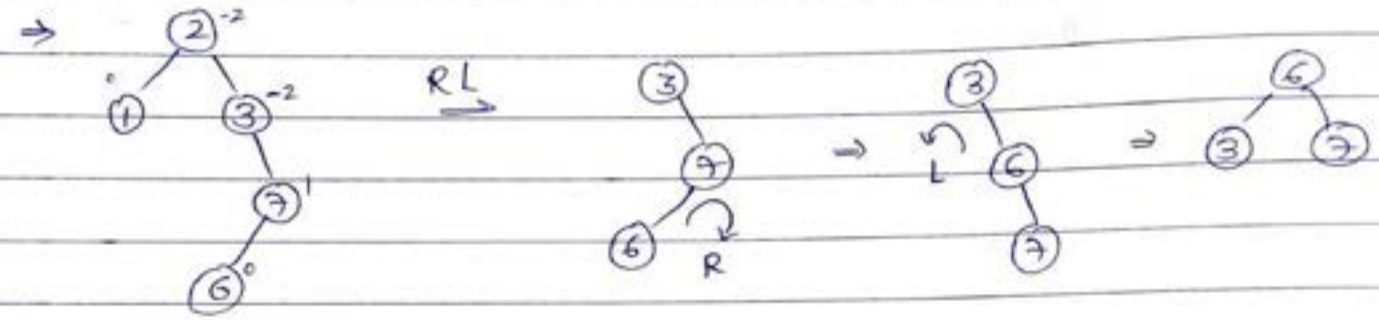


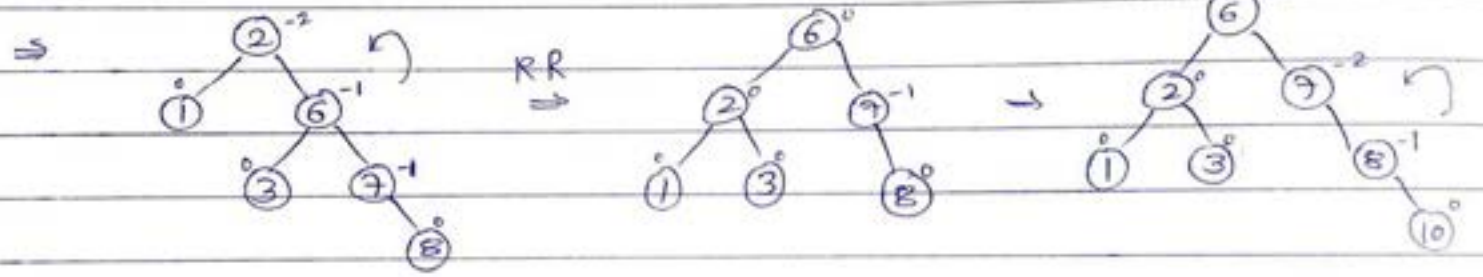
eg 3 1 2 7 6 8 10 12 11

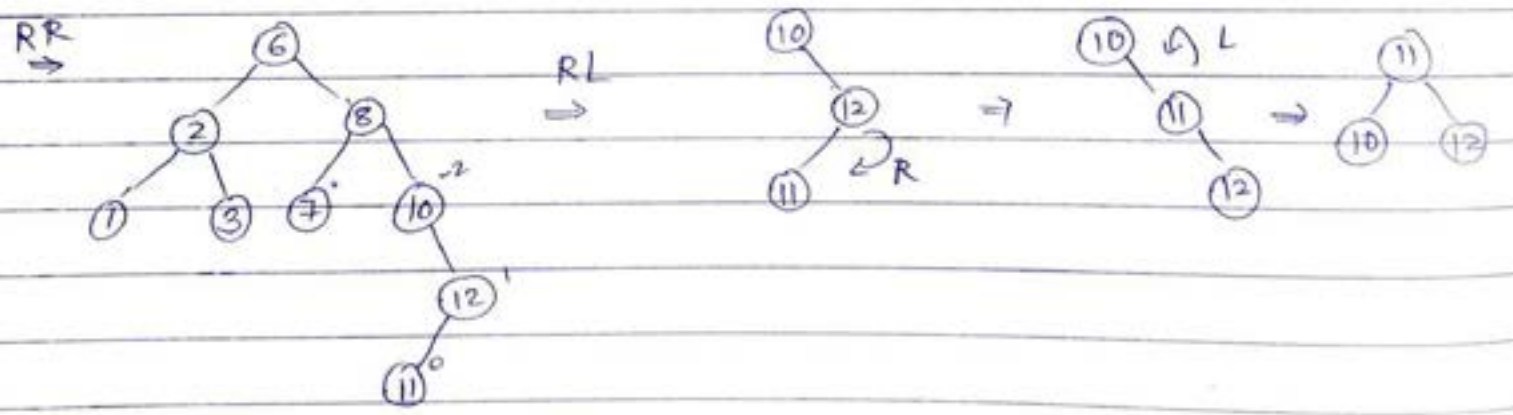


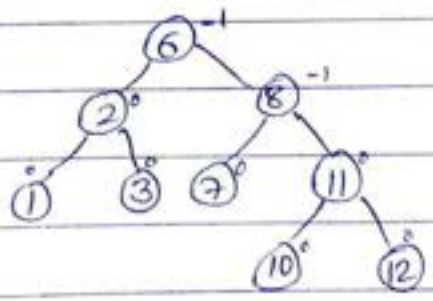
LR ⇒ (L) first rotate around child (① here)
(R) then rotate around unbalanced node (③ here)



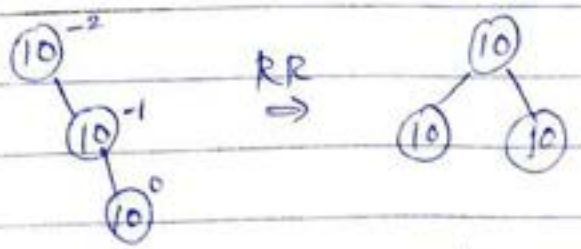








eg 10, 10, 10



Thus, we dont insert duplicate values in AVL tree.

Date: _____ Page No: _____

Complexity of AVL tree :

- worst case searching ⇒ O(logn) (always a tree, can't be a chain)
- worst case insertion: O(logn)

⇒ no. of comparisons [i.e depth (logn)] + rotations [constant time, O(1)]

⇒ O(logn)

- worst case deletion ⇒ O(logn)

* Max. nodes in an AVL tree of depth d ;

⇒ $2^{d+1} - 1$

* Min. nodes in an AVL tree of depth d ;

d = 0 ⇒ n = 1, n(0) = 1 d = 1 ⇒ n = 2, n(1) = 2 ( or)

d =

⇒ n(d) = n(d-1) + n(d-2) + 1

Que Find min no. of nodes in an AVL tree of depth 5?

n(5) = n(4) + n(3) + 1
= (n(3) + n(2) + 1) + (n(2) + n(1) + 1) + 1

Ans ⇒ 20

CS-2009

Que What is the maximum possible height of any AVL tree with 7 nodes? assume height of tree with single node is zero.

⇒ [illegible] = 3 n(3) = 7

CS-2003

Que 63

⇒
```
struct node {
int data;
struct node *left;
struct node *right;
{
void traverse (struct node *P1)
{
if (P1 != NULL)
{
traverse (P1 -> left);
printf ("%d", P1 -> data);
traverse (P1 -> right);
}
}
```

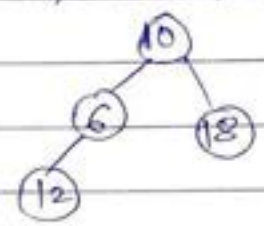
⇒ inorder (change their order for preorder & postorde

* complexity of tree traversal is Θ(n) because all the n nodes are being traversed.

Date: _____ Page No: _____

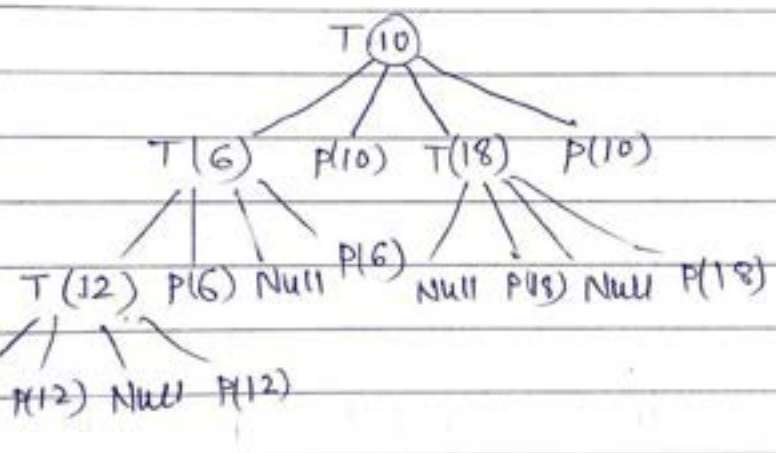
Que
```
if (P1 != NULL)
{
    traverse (P1 -> left);
    printf ("%d", P1 -> data);
    traverse (P1 -> right);
    printf ("%d", P1 -> data);
}
```

Given, tree.



P1 = 10

→



total
⇒ 9 calls

⇒ 12 12 6 6 10 18 18 10 Ans

---

* Space complexity of Tree Traversal ⇒ n + depth of tree
  - n → for tree
  - depth of tree → for stack

⇒ n + log n ⇒ O(n)

⇒ n + n (chain) ⇒ O(n)

Date:_____ Page No:____

⇒ int totalnodes (struct tree *p) // count total no. of nodes in a tree

```
{
    if (p == NULL)
        return 0;
    else
        return 1 + totalnodes (p->left) + totalnodes (p->right);
}
```

⇒ int leafnodes (struct tree *p)

```
{
    if (p->left == NULL && p->right == NULL)
        return 1;
    else
        return leafnodes (p->left) + leafnodes (p->right);
}
```

* Complexity to find smallest & largest element in ;

① AVL → O (log n) } worst
② BST → O (n) } case

⇒ int height (struct node *p1) // returns height of tree

```
{
    int l=0, r=0
    if (p1->left == NULL && p->right == NULL) return 0;
    else if (p1->left != NULL) l = height (p->left);
    if (p1->right != NULL) r = height (p->right);
    return max (l,r) + 1;
}
```

PROGRAM DESIGN TECHNIQUES

Date: Page No:

1. Greedy Method
2. Dynamic Programming
3. Divide & Conquer

* When we've to optimize a solution of a problem, greedy & dynamic prog. are used.

Greedy Method: It applies a logic or a trick by which optimal solution is found.
It assumes the logic or trick it's using always produces optimal solution.

① Fractional Knapsack Problem:

eg

	I1	I2	I3	I4
P	10	20	20	40
W	10	20	10	20

knapsack capacity ⇒ 30 kg.

⇒ Find Profit-weight ratio of all the items.

P/W	1	1	2	2

⇒ Fill the knapsack with the items having max. ratio.

	P	W
⇒ I3	20	10
I4	40	20
	60	30

Ans ⇒ 60

Brute Force ⇒ To apply all possible combinations
It always return best result but not always practically used.

Que previous que, knapsack capacity = 25

	P	W
I3	20	10
I4	40×15/20	15
		25
Ans	50	

Complexity :

⇒ $\underset{\text{p/w of all n items}}{n} + \underset{\text{sorting of p/w of all the items (descending order)}}{n\log n} \Rightarrow O(n\log n)$

→

* 0-1 knapsack problem cannot be solved by greedy but by dynamic programming.

eg

	I1	I2	I3	I4
P	19	50	19	19
W	10	25	10	10
P/W	1.9	2	1.9	1.9

Knapsack = 30 kg.

Greedy ⇒ I2 → 50 Ans ✗ But, I1, I3, I4 ⇒ 57 Ans

* Greedy fails for 0-1 knapsack problem.

Complexity ⇒ O(nlogn)

Date _____ Page No: _____

② Optimal Merge Pattern

files → no. of records

F1 → 20
F2 → 10
F3 → 15
F4 → 5

30, 45, 50

* We've to merge the files such that no. of records to be handled is minimum.

⇒ 30 + 45 + 50 = 125 ✓

F1 → 20
F2 → 10
F3 → 15
F4 → 5

20, 30, 50

⇒ 20 + 30 + 50 = 100 ✗

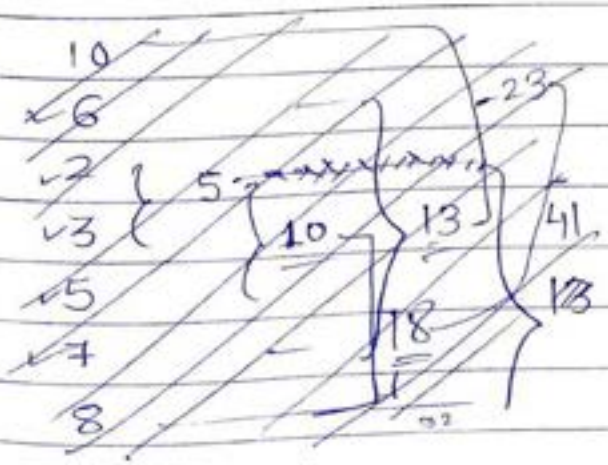
* Optimal merge pattern says "merge smaller files first."

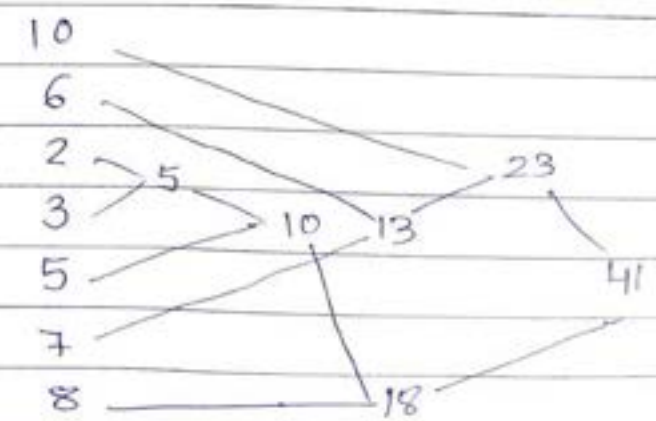
F1 → 20
F2 → 10
F3 → 15
F4 → 5

15, 30, 50

⇒ 15 + 30 + 50 = 95 ✓

Que



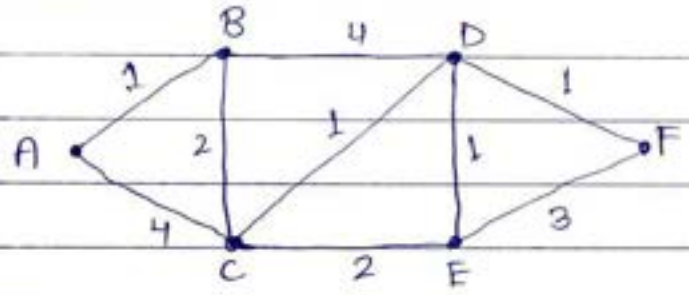


5 + 10 + 13 + 23 + 18 + 41 = 110

③ Single Source Shortest Path (Dijkstras algo)

* It works well for directed as well as undirected graphs.
* We find shortest paths from source vertex to all other vertices.

eg



Source ⇒ A

* It maintains 2 set → ① visited vertex set
② unvisited vertex set

* Visited vertex set contains the vertices whose shorter path is found out.
* Unvisited vertex set contains the vertices whose shorter path is not found out.

* If distance is same, select the previous one. (as it is)

Date: _____ Page No. _____

	Visited	Distance	through
A	✓	0	
B		∞	
C		∞	
D		∞	
E		∞	
F		∞	

* No. of loops = no. of vertices

①

V		D	T
A	✓	0	
B		1	A
C		4	A
D		∞	
E		∞	
F		∞	

②

V		D	T
A	✓	0	
B	✓	1	A
C		3	B
D		5	B
E		∞	
F		∞	

③

V		D	T
A	✓	0	
B	✓	1	A
C	✓	3	B
D		4	C
E		5	C
F		∞	

④

V		D	T
A	✓	0	
B	✓	1	A
C	✓	3	B
D	✓	4	C
E		5	C
F		5	D

⑤

V		D	T
A	✓	0	
B	✓	1	A
C	✓	3	B
D	✓	4	C
E		5	C
F	✓	5	D

(equal → take any)

⑥

V		D	T
A	✓	0	
B	✓	1	A
C	✓	3	B
D	✓	4	C
E	✓	5	C
F	✓	5	D

Order ⇒ ABCDFE

⇒

	V	D	T
A	✓	0	
B	✓	1	A
C	✓	3	B
D	✓	4	C
E	✓	5	C
F	✓	5	D

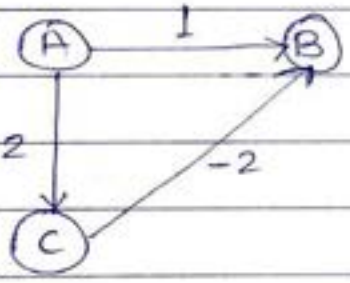
length of
• (A) Shortest path from A to F
⇒ 5

• Shortest path from A to F
⇒ FDCBA or ABCDF

• Shortest path from A to E ⇒ ECBA or ABCE

CS. 2004 Que 44

*



⇒

①	V	D	T
A	✓	0	
B		1	A
C		2	A

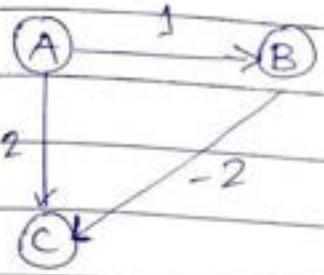
②	V	D	T
A	✓	0	
B	✓	1	A
C		2	A

③	V	D	T
A	✓	0	
B	✓	1	A
C	✓	2	A

* It returns shortest distance from A to B is 1, but it should be 0. (ACB). Thus, it fails here.

*

Date _____ Page No: ____

* 

⇒ ①

	V	D	T
A	✓	0	
B		1	A
C		2	A

②

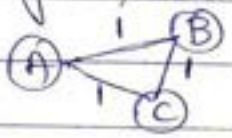
	V	D	T
A	✓	0	
B	✓	1	A
C		-1	B

③

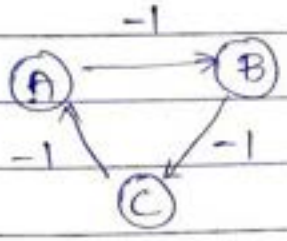
	V	D	T
A	✓	0	
B	✓	1	A
C	✓	-1	B

⇒ Now, it doesn't fail here.

* If there exists a cycle of positive edges, then there exists no longest path.



* If there exists a cycle of negative edges, then there exists no shortest path. (dijkstra's algo always fails)

eg 

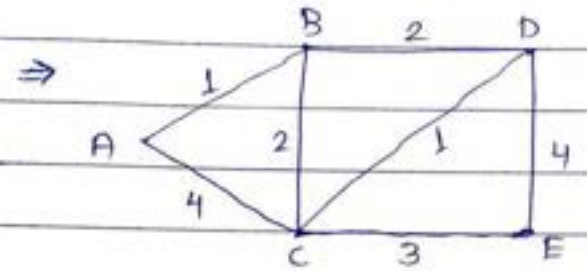
	V	D	T
A	✓	0	
B	✓	-1	A
C	✓	-2	B

fail

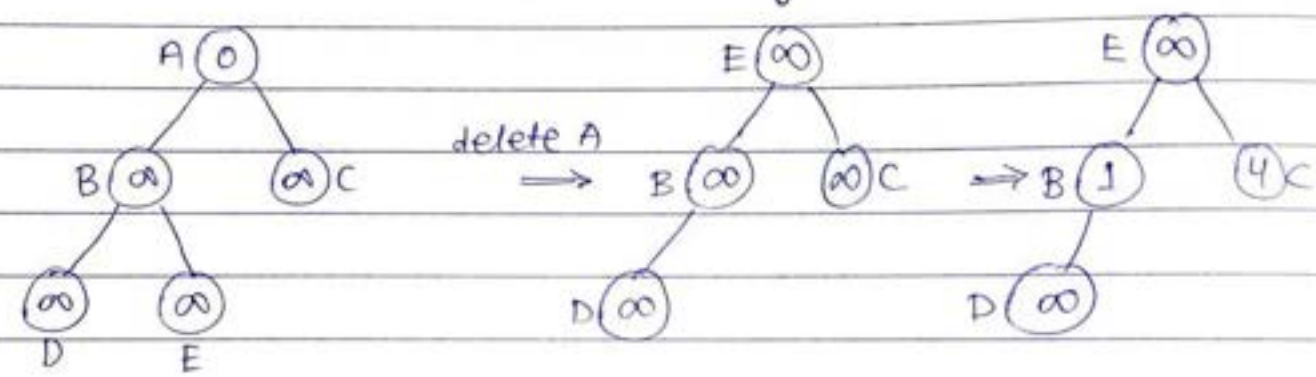
* But, if some are positive & some are negative edges, dijkstra's algo may work well or may fail.

* It returns diff. complexity with diff data structures we're discussing with heap.

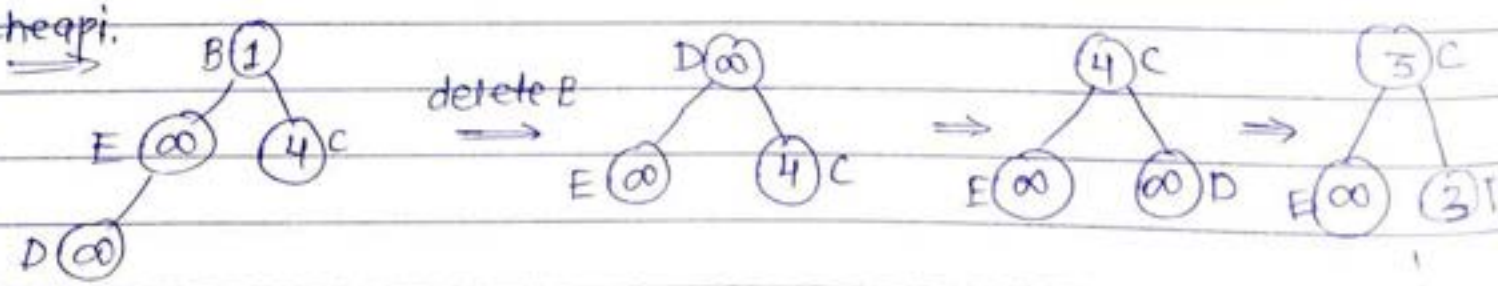
Date: Page No:



* It creates min heap. internally to store distances



* Min heap contains all unvisited vertices & as they are visited, they are deleted from min heap.



⇒ Complexity ⇒ create heap + Delete all the vertices one by one

* If performs edge relax. from unvisited vertices only

+ Edge relaxation [O(1)] (then reheapification)

* BFS is best algo to find shortest path in unweighted graph ⇒ $O(V+E)$

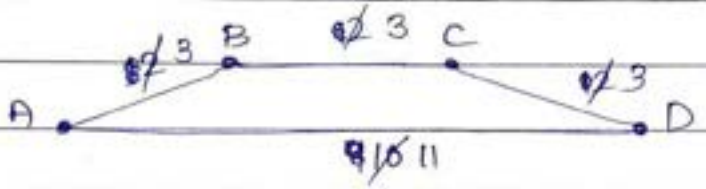
Date:_____ Page No:_____

$$\Rightarrow V + V\log V + E * 1 * \log V$$

$$\Rightarrow V\log V + E\log V \Rightarrow O[(V+E)\log V]$$

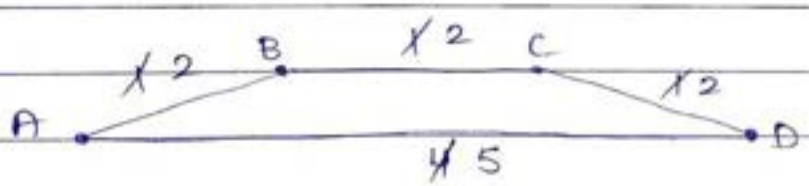
* If weight of every edge is increased by a common factor, shortest path may or may not be changed but min. spanning tree will remain same.

eg



• shortest path
⇒ ABCD (old)
(+1) ⇒ ABCD (new)
(same)

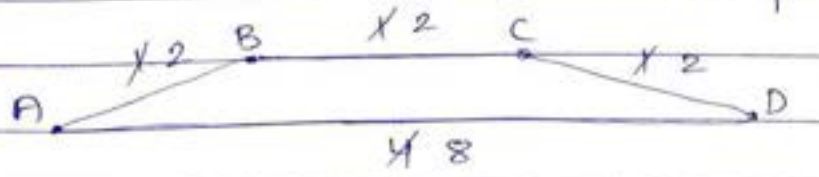
eg



⇒ ABCD (old)
(+1) ⇒ AD (new)
(change)

* If weight of every edge is multiplied by a constant factor, shortest path will remain same. & min spanning tree will also be same.

eg



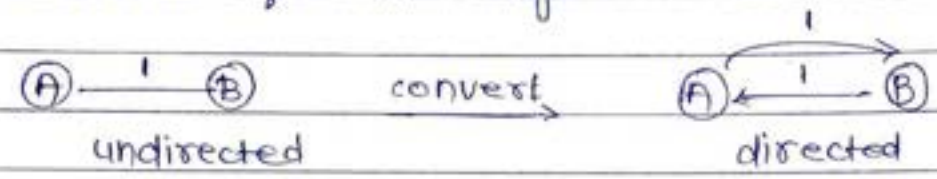
⇒ ABCD (old)
(*2) ⇒ ABCD (new)
(same)

Date: _____ Page No: _____

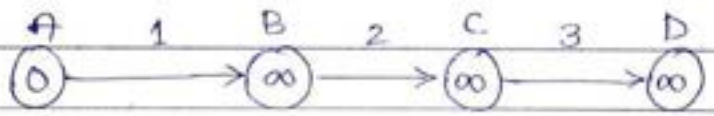
Single Source Shortest Path (Bellman - Ford Algorithm)

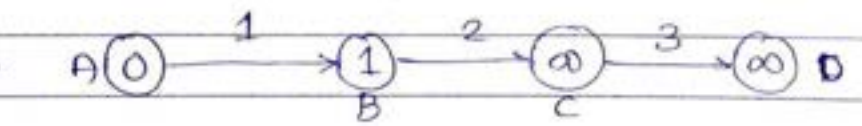
* It's not a problem of greedy but of dynamic prog.

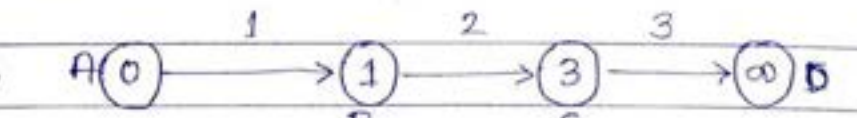
* It works well for negative edges but graph must be directed for this algo.

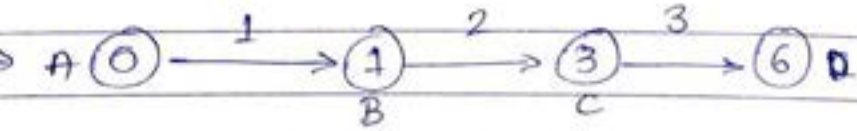
→ 

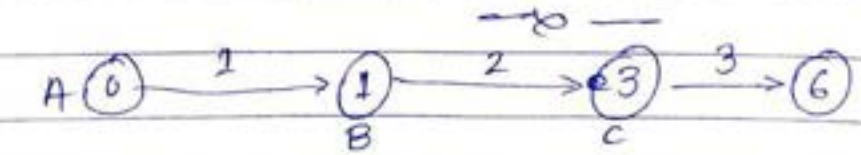
* It performs edge relaxation from every vertex in every iteration

eg 

(1) → 

(2) → 

(3) → 

(4) ⇒ 

same

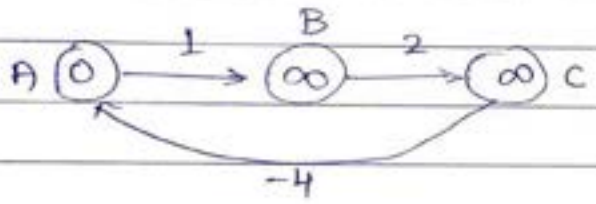
Date: _____ Page No: _____

* In (n-1) iterations, for n vertices, shortest path will be finalised in worst case.

* In n^{th} iteration, there will be no change in the path.

* If there exists a cycle of negative edges, $(n-1)^{th}$ & n^{th} iteration are not same then it will return error. that there exists no shortest path.

eg

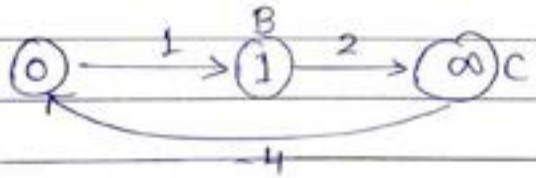


→ cycle of -ve edge

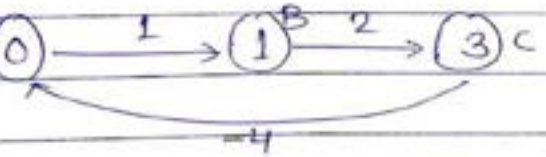
⇓

$1 + 2 + (-4) = -1$ (-ve)

(1) ⇒

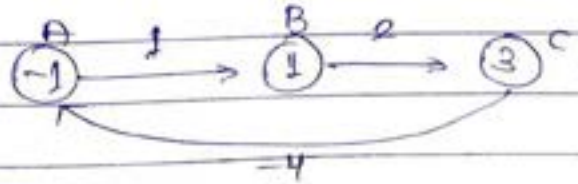


(2) ⇒



not same ⇒ error

(3) ⇒



• there exists no shortest path.

- Complexity → $\Theta(V \cdot E)$ // loop runs for V times & in each iteration, E edges are relaxed.

Date: _____ Page No: _____

(4) Huffman / Prefix Code: To minimize length of codewords or messages.

* Generally, a character is represented by 8 bits. But, in alphabets, vowels are more frequent than consonants in words. Thus, to minimize the length of message, use short-length codeword for more frequent characters & long codewords for less freq. characters.

⇒ eg
a → 0
e → 1
i → 00

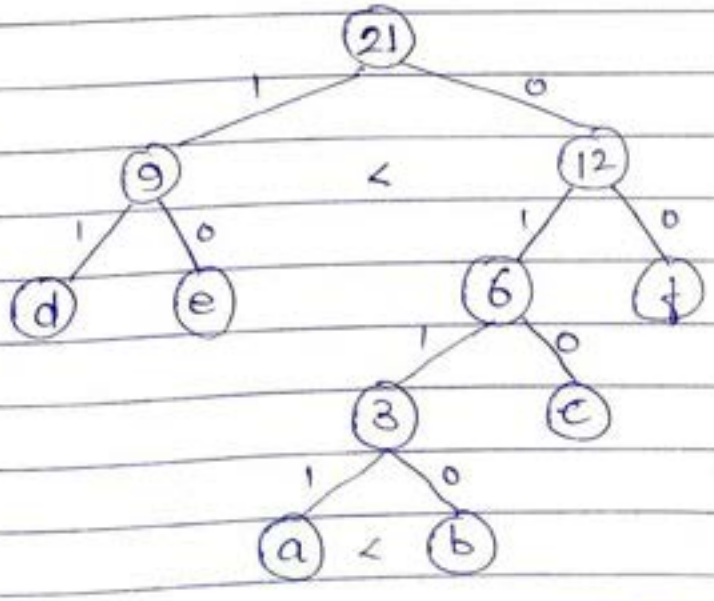
* If 00 is sent ⇒
(a) aa or i ?

⇒ Thus, no codeword should be prefix of any other codeword; for this, huffman tree is used.

eg

a	b	c	d	e	f
1	2	3	4	5	6

freq. ⇒



⇒ 1f000111
⇓
dfa

Date: _____ Page No: ___

CS.2007

Que 76, 77

sorting ↓ tree creation + reading codes
$n \log n$ n $\log n$

- Complexity ⇒

Date: Page No:

⑤ Job Scheduling with deadline:

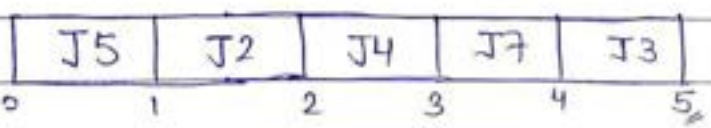
* We've to have maximum profit

Job	Profit	Deadline
J1	2	3
J2	6	2
J3	8	5
J4	7	4
J5	5	4
J6	3	1
J7	8	5
J8	5	2

* every job takes 1 unit of time for execution.

* If job is completed within deadline, then only you'll get profit.

⇒



5 + 6 + 7 + 8 + 8 ⇒ 32

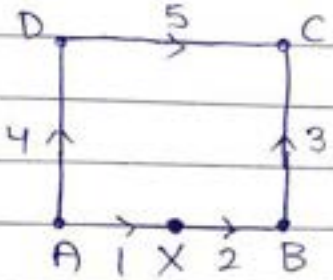
* It's an NP-complete problem ; but for fixed no of tasks/jobs, it can be solved.

Date: _____ Page No: _____

Dynamic Programming:

- It can solve the problems which have 2 properties;
  (a) Optimal substructure
  (b) Overlapping subproblem

(a) Optimal solution of the main problem exists in the optimal solution of the subproblems.

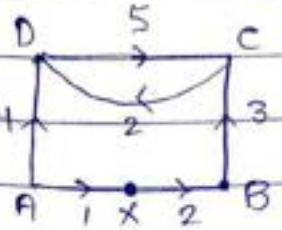
eg 

∴ find shortest path from A to C.
given, that, this shortest path contains X.

∴ Subproblems ⇒ ① Shortest path from A to X
② Shortest path from X to C

⇒ Thus, it exhibits optimal substructure.

eg ; find longest path from A to C in above graph.



assume, cycles can't be repeated.

- Subproblems ⇒ ① Longest path from A to D
② Longest path from D to C

- A to D → AXBCD , D to C → DC
∴ A to C → AXBCDC ✗ it doesn't exist in the graph.

⇒ Thus, it doesn't exhibit optimal substr.

(b) Overlapping Subproblem:

```
int fib(int n)
{
  if (n==1 || n==2)
    return 1;
  else
    return fib(n-1) + fib(n-2);
}
```

fib(5) → fib(4) + fib(3)
problem subproblems

fib(4) → fib(3) + fib(2)

* When same subproblem is used to solve many subproblems.
eg fib(3) is used to solve fib(5) & fib(4)

* Here, fib(3) is getting called twice; by fib(5) & fib(4).

* But, dynamic program won't call it twice but only once & save it to some data structure; & it's known as memorization (to store results of subproblems in some data structures)

* In quick sort, we divide the problem into subproblems but the subproblems are not used again, thus it's not a problem of dynamic programming.

Date: Page No:

* There can be 2 approaches of dynamic programming.

① Bottom-up

fib(2)
↑
fib(1)
↑
fib(0)

(starts from smaller subproblem)

② Top-down (starts from larger subproblem)

fib(2)
↓
fib(1)
↓
fib(0)

(1) Matrix Chain Multiplication Problem :

- $A_{3\times2}$ $B_{2\times1}$ $C_{1\times3}$ $D_{3\times2}$ → matrix chain

* We've to multiply all of them.

eg $(AB)C = A(BC)$

* matrix multiplication is associative but not commutat

* $A_{m\times n}$ $B_{n\times k}$ ⇒ to multiply these 2, min. no. of multiplications is $(m*n*k)$

⇒ ① $(AB)_{3\times1}$ ⇒ $3*2\times1 = 6$ multiplications

$((AB)_{3\times1} C)_{3\times3}$ → $3\times1*3 = 9$

$(ABC)D)_{3\times2}$ → $3*3\times2 = 18$

total multiplications ⇒ $6+9+18 = \underline{33}$

$(((AB)C)D)$

Date: Page No:

② ⇒ AB → 3.2.1 = 6

CD ⇒ 1.3.2 = 6

((AB)(CD)) ⇒ 3.1.2 ⇒ 6

→ 6+6+6 = 18

③

* There exists 5 ways to multiply 4-matrix chain.

Brute Force

① (((AB)C)D) ⇒ 33

② ((AB)(CD)) ⇒ 18 ✓ min (here)

③ ((A(BC))D) ⇒ 42

④ (A((BC)D)) ⇒ 30

⑤ (A(B(CD))) → 22

* If chain is of more than 4 matrices, then there will be many combinations & it will take much time.

Date: Page No:

⇒ By dynamic programming.

A1	A2	A3	----	An
$P_0 \times P_1$	$P_1 \times P_2$	$P_2 \times P_3$		$P_{n-1} \times P_n$

→ min. no of multiplications, $m[1, n]$

① $A_1 (A_2 -- A_n)$

② $(A_1 . A_2)(A_3 - - A_n)$

③ $(A_1 . A_2 . A_3)(A_4 - - A_n)$

⋮

$(A_1 A_2 . - - A_{n-1}) A_n$

→ minimum

For ① $A_1 (A_2 -- A_n) \Rightarrow m[1,1] + m[2,n] + P_0 * P_1 * P_n$

$m(2,n) =$ ① $A_2 (A_3 -- A_n)$

② $(A_2 . A_3)(A_4 . -- A_n)$

⋮

→ minimum

eg

⇒ ② $(A_1 . A_2)(A_3 - - - A_n) \Rightarrow m[1,2] + m[3,n] + P_0 * P_2 * P_n$

$P_0 * P_2$ $P_2 * P_n$

$m[i,j]$ ⇒ min. no. of multi required to multiply chains A_i to A_j.

Date:_____ Page No:_____

Also, ③ $(A_1 A_2 A_3)(A_4 \dots A_n) \Rightarrow m[1,3] + m[4,n] + P_0 * P_3 * P_n$

④ $(A_1 A_2 A_3 \dots A_{n-1}) A_n \Rightarrow m[1,n-1] + m[n,n] + P_0 * P_{n-1} * P_n$

$$\Rightarrow m[1,n] = \min_{1 \le k < n} \{ m[1,k] + m[k+1,n] + P_0 * P_k * P_n \}$$

OR

$$\Rightarrow m[i,j] = \min_{i \le k < j} \{ m[i,k] + m[k+1,j] + P_{i-1} * P_k * P_j \}$$

- $m[i,j] = 0 \quad ; \quad i = j$

eg ~~$A_{1\,3\times1}$ $B_{1\times2}$ $C_{2\times3}$ $D_{3\times2}$~~ $A1_{3\times1}$ $A2_{1\times2}$ $A3_{2\times3}$ $A4_{3\times2}$

⇒ $i \le j$ (i can't be greater than j)

$i=4$				0
$i=3$			0	12
$i=2$		0	6	12
$i=1$	0	6	15	18
	$j=1$	$j=2$	$j=3$	$j=4$

→ these diagonals can be computed directly

- Bottom Up (smaller to larger subproblems).

$$m[1,2] = \min_{1 \le k < 2} \{ m[1,k] + m[k+1,2] + P_0 * P_k * P_2 \}$$

$k=1 \Rightarrow \min \{ m[1,1] + m[2,2] + 3 \times 1 \times 2 \} = \underline{6}$

Date: _____ Page No: _____

$m[2,3] = \min_{2 \le k < 3} \{ m \}$

$k = 2 \Rightarrow$

$m[2,3] = 1 \times 2 \times 3 = 6$

$m[3,4] = 2 \times 3 \times 2 = 12$ } directly

- $m[1,3] = \min_{1 \le k < 3} \{ m[1,k] + m[k+1,3] + P_0 * P_k * P_3 \}$

$k = 1 \Rightarrow \quad m[1,1] + m[2,3] + 3 \times 1 \times 3 = 0 + 6 + 9 = 15$

$k = 2 \Rightarrow \quad m[1,2] + m[3,3] + 3 * 2 * 3 = 6 + 0 + 18 = 24$

$\therefore \quad m[1,3] = \min(15, 24) = 15$

- $m(2,4) = \min_{2 \le k < 4} \{ m[2,k] + m[k+1,4] + P_1 * P_k * P_4 \}$

$k = 2 \Rightarrow \quad m[2,2] + m[3,4] + 1 * 2 * 2 = 0 + 12 + 4 = 16$

$k = 3 \Rightarrow \quad m[2,3] + m[4,4] + 1 * 3 * 2 = 6 + 0 + 6 = 12$

$\therefore \quad m[2,4] = \min(16, 12) = 12$

- $m(1,4) = \ldots = 18$

Date: Page No:

(2) Subset Sum Problem:

$A = \{2, 3, 1, 5, 4\}$; a set is given

* Is there a subset whose sum / weight is k (constant integer).

* It's an NP-complete problem but it can be solved only for +ve integers set.

* There are (n+1) rows in the table & (K+1) columns where n is no. of elements in set & k is sum. of subset.

eg $A = \{3, 2, 1, 5, 4\}$ (a1 a2 a3 a4 a5)

• Bottom Up

⇒

j →	0	1	2	3	4	5	6	7	8	9	10
i = 0	T	F	F	F	F	F	F	F	F	F	F
1	T	F	F	T	F	F	F	F	F	F	F
2	T	F	T	T	F	T	F	F	F	F	F
3	T	T	T	T	T	T	T	F	F	F	F
4	T	T	T	T	T	T	T	T	T	T	T
5	T	T	T	T	T	T	T	T	T	T	T

starting i^{th} elements of A has a subset of sum = j ⇒ True

Date: Page No.

- $M[i,j] = F \quad ; \quad i = 0 \ \& \ j \neq 0$
- $M[i,j] = T \quad ; \quad j = 0$
- $M[i,j] = M[i-1, j] \underset{OR}{\vee} M[i-1, j-a_i]$

* Complexity → n+1 rows * k+1 columns = total cells

→ $\Theta(nk)$

eg $\underset{a_1\ a_2\ a_3\ a_4}{(1, 2, 3, 4)}$

⇒ $M(4,6) = M(3,6) \vee M(3, 6-4)$

$M(3,2)$

Date: _____ Page No: _____

③ Longest Common Subsequence :

X & Y are 2 strings.

X = "ABBACBCA" , Y = "BABAACCB"

Here, "BBA" is a substring of X but not of Y.
but "BBA" is a subsequence of Y & X.

"BCA" is not a subsequence of Y because reverse traversal is not allowed. (only left to right)

⇒ We've to find the longest common subsequence b/w 2 strings.

Eg $X = X_1 X_2 X_3 X_4 \ldots X_n$, $Y = Y_1 Y_2 Y_3 Y_4 \ldots Y_m$

⇒ There are $n+1$ rows & $m+1$ columns.

	0	Y_1	Y_2	Y_3	. . .	Y_m
0						
X_1						
X_2						
X_3						
!						
!						
!						
X_n						

- $M[i,j] = 0$ if $i=0$ or $j=0$
- $M[i,j] = \max[M[i-1,j], M[i,j-1]]$ if $x_i \neq y_j$
- $M[i,j] = M[i-1, j-1] + 1$ if $x_i = y_j$

eg $X =$ "ABBCBA" , $Y =$ "BACBAB"

	j →	0	1 B	2 A	3 C	4 B	5 A	6 B
⇒	0	0	0	0	0	0	0	0
i =	1 A	0	← 0	↖ 1	← 1	← 1	↖ 1	← 1
	2 B	0	↖ 1	← 1	← 1	↖ 2	← 2	↖ 2
	3 B	0	↖ 1	← 1	← 1	↖ 2	← 2	↖ 3
	4 C	0	↑ 1	← 1	↖ 2	← 2	← 2	↑ 3
	5 B	0	↖ 1	← 1	↑ 2	↖ 3	← 3	↖ 3
	6 A	0	↑ 1	↖ 2	← 2	↑ 3	↖ 4	← 4

length of longest common subsequence is 4.

⇒ ~~X = "ABBCBA"~~

Y = "BACBAB"

X = "ABBCBA"

⇒ BCBA (←)

take cell with ↖ diagonal arrow

* we can also compute it column-wise — ?

* there can be multiple longest common subsequences

Date: _____ Page No: _____

* Complexity → $m * n$ → $\Theta(m * n)$

Date: _____ Page No: _____

Another Quick Sort (from cormen) Algo.

* Here, we select last element as pivot. & only left to right scan is done.

	2	7	8	3	6	1	4
i = -1	0	1	2	3	4	5	6
	↑ j						↑ pivot

⇒ Incre. i, when we find an element smaller than pivot. & swap a[i] & a[j]. & incre. j.

⇒ 2 7 8 3 6 1 4 i = ~~-1~~ 0

~~↑ j~~ ~~↑ j~~ ~~↑ j~~ ~~↑ j~~ ↑ pivot

Now, incre. i = ~~0~~ 1 & swap a[i] & a[j] & incre. j

↓

a[1] & a[3]

⇒ 2 3 8 7 6 1 4 i = ~~1~~ 2

~~↑ j~~ ↑ j ↑ pivot

• swap a[2] & a[5] & i

⇒ 2 3 1 7 6 8 4 i = ~~2~~ 3

↑ j ↑ pivot

incre i, Now, swap a[i] & pivot → a[2] & pivot

→ 2 3 1 4 7 6 8 7

Date: Page No:

Que 10 12 5 1 6 13 8 15 12 13 9

i = -1 0

i = -1 0 1 ⇒ 5 12 10 1 6 13 8 15 12 13 9

i = -1 0 1 2 ⇒ 5 1 10 12 6 13 8 15 12 13 9

i = -1 3 ⇒ 5 1 6 12 10 13 8 15 12 13 9

i = -1 4 ⇒ 5 1 6 8 10 13 12 15 12 13 9

i = -1 5 ⇒ 5 1 6 8 3 13 12 15 12 10 9

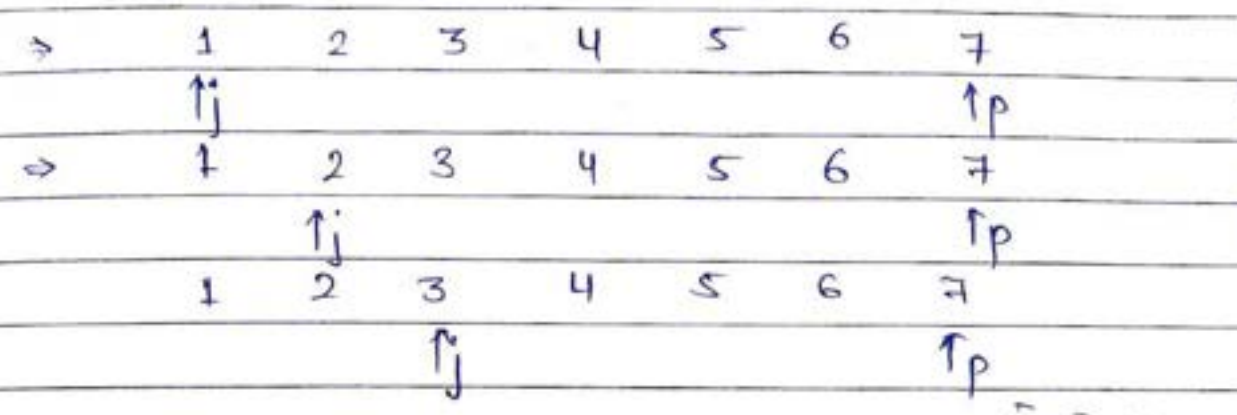
⇒ 5 1 6 8 3 9 12 15 12 10 13

smaller than pivot — greater than pivot

Now, do the same for the sublists

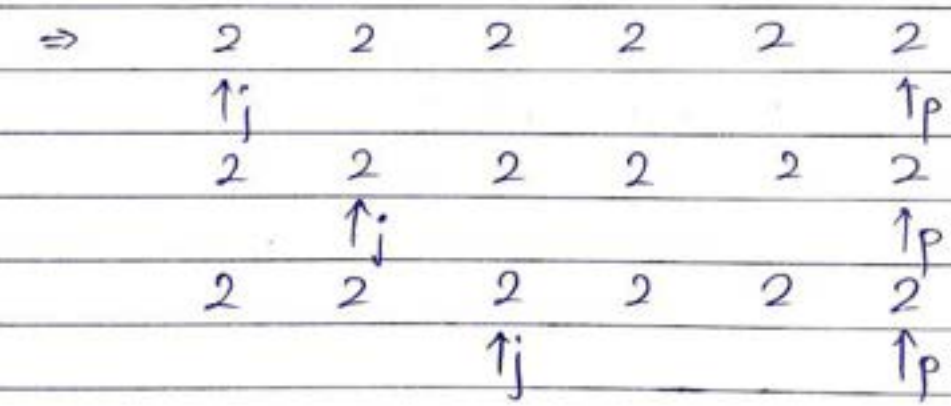
⇒ Algorithm of Quick Sort:

```
int partition (int a[], int left, int right)        // O(n)
{
    int i = left-1 , j= left , pivot = a[right];
    for ( ; j<=right-1 ; j++)
    {
        if (a[j]<= pivot)
        { i++; swap (a[i], a[j]); }
    }
    i++; swap ( a[i] , a[right]);
    return i;
}

void quicksort (Int a[], int left ,int right)
{
    if (left< right)
    { int i = partition (a, left, right);
      quicksort (a, left , i-1);
      quicksort (a, i+1, right);
    }
}

Void main()
{  int a[] = { 10, 16, 5, 9, 12, 13, 3 };
   quicksort (a, 0, 6);
}
```



No. of comparisons = $(n-1) + (n-2) \ldots\ldots 1$

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



Thus, worst case of quicksort is when array is already sorted or when list is not getting divided into 2 parts.

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

$\Rightarrow O(n^2)$

⇒

* The best case of quick sort is when array is divided into 2 nearly equal parts.

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

Thus, when list is getting divided in 2 parts (not necessarily 2 equal parts), complexity will be $O(n\log n)$.

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

⇒ $\frac{n}{3} + \frac{2n}{3} = \frac{3n}{3} = n = \theta(n)$

$\therefore$ ⇒ $O(n\log n)$