"Algorithms"

Q:- What is an Algorithm and Program?

Ans:- Step by step procedure for solving a computational problem and Program also the same meaning.

So what is difference :-

Algorithm [Priori Analysis]	Program [Posteriori Testing]
(i) In SDLC, Design perform just you include rough work. You can write on paper not proper syntax.	(i) Implementation means write the code which execute and remember proper syntax.
(ii) Domain Knowledge	(ii) Programmer
(iii) Any Language :- English like Language, some mathematical Language.	(iii) Programming Language :- C, C++, Java, Python.
(iv) Not dependent on H/w & S/w	(iv) Depedent on H/w & S/w.
(v) After writing algorithm, we analysing will complete within time and space.	(v) Testing :- Watch time and space.

Characteristics of Algorithm :-

1. **Input :-** It has 0 or more inputs.
2. **Output :-** It gives atleast one O/P.
3. **Definiteness :-** Every statement has understood and only one meaning.
4. **Finiteness :-** Algorithm must terminated or finite steps to stop either thousand of statements or lakh statements.
5. **Effectiveness :-** you should not write unnecessary statements. For chemistry also write some laboratory for chemistry.

Problem
↓
Program in C

Before program written we first design algorithm (solution)

P, [Problem]

(Design) [A1 A2 A3 A4 ....]

we have number of solutions to solve any particular problem

we will see and find out which one is better depends two parameter :-

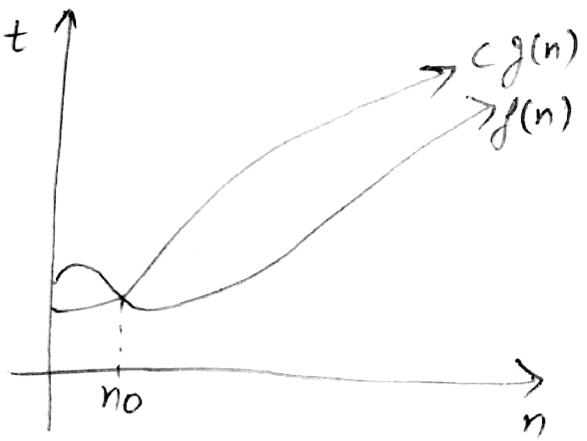
(i) Time (ii) Space / Memory] Analysing

→ Main aim is less time & less memory.

Before going to design & analysis. First familiar with some notations.

Asymptotic Notation :-

(i) Big (oh) O :-



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

$n \geq n_0$

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

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

$\leq$ (1) Bigoh (O)

$\geq$ (2) Big omega (Ω)

$=$ (3) Theta (Θ)

$<$ (4) small oh (o)

$>$ (5) Small omega (ω)

For ex :-

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

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

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

$3n+2 \leq c.n$ if $c = 4$

$3n+2 \leq 4n$

It will satisfy $n \geq 2$

Note :- $f(n) = 3n+2$, $g(n) = n$

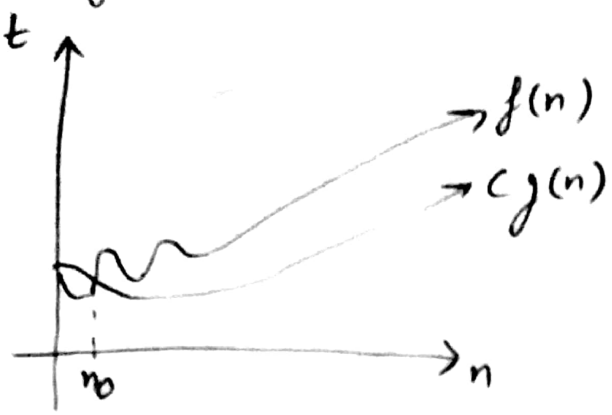
It always satisfy $f(n) = O(g(n))$

It means it will satisfy all upper value but always we count least upper bound [Tightest bound]

$g(n) = n^2$
n^3
n^n
2^n

(ii) Big Omega (Ω) :-

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



For ex:-

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

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

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

$3n+2 \geq c.n$ $\quad c=1$, $n_0 \geq 1$

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

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

Is $3n+2 = \Omega(n^2)$? (No)

$3n+2 \geq c.n^2$

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

c_0 for tightest lower bound

$\log n$

$\log \log n$

Ex: $f(n) = n^2$, $g(n) = n^2+n$

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

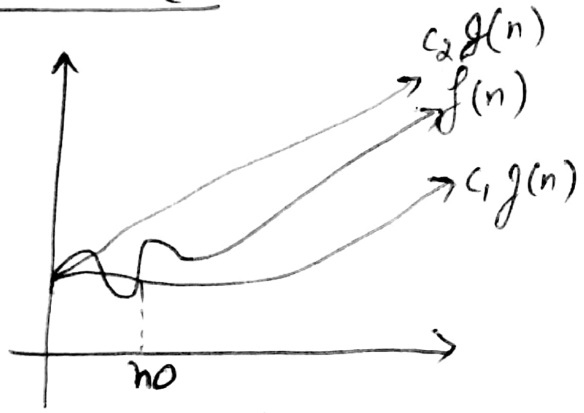
$n^2 \geq c.(n^2+n)$ $\quad c = \frac{1}{2}$

$n^2 \geq \frac{1}{2}n^2 + \frac{1}{2}n$

$\frac{1}{2}n^2 \geq \frac{1}{2}n$

$n \geq 0$

(iii) Big theta (Θ) :-



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

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

$c_1, c_2 > 0$, $n \geq n_0$

$n_0 \geq 0$

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

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

$f(n) \leq c\,g(n)$

$3n+2 \leq 4n$

$n_0 \geq 1$, $c = 4$

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

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

$n_0 \geq 1$

we have another

$3n^2+n+1 = \Theta(n^2)$

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

$f(n) = \Omega\, g(n)$ Cond1: $f(n) \geq c_1 g(n)$

$f(n) = O(g(n))$ Cond2: $f(n) \leq c_2 g(n)$

O [Big-oh]	Ω [Big-omega]	Θ [Theta]
Worst Case	Best Case	Average Case
Upper bound	Lower bound	Average.

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

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

$n^2+n = O(5n^2)$

$n^2+n \leq c_2 (5n^2)$ $(c_2 = 1)$

$5n^2 \geq n^2+n$

$4n^2 \geq n$ $\quad n \geq 0$ ↓ n_0

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

$n^2+n \geq c_1 \cdot (5n^2)$ $(c<1)$

$n^2+n \geq \frac{1}{5}(5n^2)$ $(c_1 = \frac{1}{5})$

$n^2+n \geq n^2$

$n \geq 0$

Another example :-

$\Rightarrow$ $f(n) = n^2 + 5n + 100$ $\quad g(n) = n^2$

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

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

$n^2+5n+100 \leq c \cdot n^2$ $\quad c = 106$

$n^2+5n+100 \leq 106n^2$ $\quad n = 0$

$\Rightarrow$ $f(n) = n^2$ $\quad g(n) = n^2+5n+100$

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

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

$n^2 \geq c \cdot (n^2+5n+100)$ $(c = 1/2)$

$n^2 \geq \frac{1}{2}n^2 + \frac{5}{2}n + 50$

$\frac{1}{2}n^2 \geq \frac{5}{2}n + 50$

$n^2 \geq 5n + 100$ $\quad (n \geq 20)$

④ Small oh (o) :- (<)

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

Ex :- $f(n) = n$, $g(n) = n^2$

$$\lim_{n\to\infty} \frac{n}{n^2} = \frac{1}{n} = \frac{1}{\infty} = 0$$

Ex :- $f(n) = \log n$ $\quad g(n) = n$

$$\lim_{n\to\infty} \frac{\log n}{n} = \frac{\log \infty}{\infty} = \frac{\infty}{\infty}$$ Indefinite forms

Also, $\frac{0}{0}$ Indefinite forms

Then apply L'Hospital rule :-

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

$$\lim_{n\to\infty} \frac{1/n}{1} = \frac{1}{\infty} = 0$$

$\log n = o(n)$

⑤ Small omega (ω) :- (>)

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

Ex :- $f(n) = 3^n$, $g(n) = 2^n$

$$\lim_{n\to\infty} \frac{3^n}{2^n} = \left(\frac{3}{2}\right)^n = \left(\frac{3}{2}\right)^\infty = (1.5)^\infty = \infty$$

Ex :- $f(n) = n^2$, $g(n) = \log n$

$$\lim_{n\to\infty} \frac{n^2}{\log n} = \frac{\infty^2}{\log \infty} = \frac{\infty}{\infty}$$

Apply L'Hospital rule

$$\lim_{n\to\infty} \frac{2n}{1/n} = \lim_{n\to\infty} 2n^2 = 2(\infty)^2 = 2\infty = \infty$$

So, $n^2 = \omega(\log n)$

How to write an algorithm :-

Algorithm Swap (a, b)

{

Every statement takes 1 time

temp = a ; —— 1

a = b ; —— 1

b = temp ; —— 1

}

$f(n) = 3$

No need in case algorithm

(i) Data type

(ii) No declaration of any variable

Uses :-

(i) { } or ~~begin~~ begin end

(ii) = or := or ←

Criteria for analysing an algorithm :-

1. Time } Main criteria
2. Space }
3. N/w congestion :- How many data transfer.
4. Power consumption
5. CPU registers :- How many registers it is used.

(2) Space :- What are variables used in algorithm.

a - 1

b - 1

temp - 1

$S(n) = 3$ words $O(1)$

(1) Time :- $f(n) = 3$ $O(1)$

Frequency count method :-

Find sum of all the elements in array

Algorithm Sum (A, n) — A: Array, n: no. of elements

```
{
    s = 0;                          ---------- 1
    for (i = 0; i < n; i++)         ---------- n+1
    {                                  1   n+1   n
        s = s + A[i];               ---------- n
    }
    return s;                       ---------- 1
}                                   f(n) = 2n+3   O(n)
```

Space Complexity :-

A - n
n - 1
s - 1
i - 1

S(n) = n+3

Degree is n
O(n)

Sum of two matrices :-

Space :- A - n^2
B - n^2
C - n^2
n - 1
i - 1
j - 1

$S(n) = 3n^2 + 3$
$O(n^2)$

Algorithm Add (A, B, n)

3×3
n×n

```
{
    for (i = 0; i < n; i++)             ---------- n+1
    {
        for (j = 0; j < n; j++)         ---------- n × (n+1)
        {
            C[i,j] = A[i,j] + B[i,j];   ---------- n × n
        }
    }
}
```

$f(n) = 2n^2 + 2n + 1$
$O(n^2)$

Multiplication of two matrices :-

```
Algorithm Multiply (A, B, n)
{
  for (i=0 ; i<n; i++)          ——————— (n+1)
  {
    for (j=0; j<n; j++)         ——————— (n+1) n
    {
      c[i,j] = 0;               ——————— n × n
      for (k=0; k<n; k++)       ——————— (n+1) × n × n
      {
        c[i,j] = c[i,j] + A[i,k] * B[k,j]; —— n × n × n
      }
    }
  }
}
```

$f(n) = 2n^3 + 3n^2 + 2n + 1$

Degree of polynomial $O(n^3)$

Space :-

A — n^2
B — n^2
C — n^2
n — 1
i — 1
j — 1
k — 1

$S(n) = 3n^2 + 4$

$O(n^2)$

Time Complexity

(1)
```
for ( i = 1 ; i<n ; i++)
{
  stment ;  ——— n
}
```
$O(n)$

(2)
```
for (i = n ; i < 0 ; i--)
{
  stmt ;  ——— n
}
```
$O(n)$

(3)
```
for (i = 1 ; i<n ; i = i+2)
{
  stmt ;  ——— n/2
}
```
$O(n)$

(4)
```
for (i = 1 ; i<n ; i = i+20)
{
  stmt ;  ——— n/20
}
```
$O(n)$

n by anything is $O(n)$

(5)
```
for (i=0; i<n; i++ ) ——— (n+1)
{
  for (j=0 ; j<n; j++) ——— n × (n+1)
  {
    stmt ;  ——————— n × n
  }
}
```
$O(n^2)$

(6)
```
for (j=0; i<n; i++)
{
  for (j=0; j<i; j++)
  {
    stmt ;
  }
}
```

i	j	no. of stmt
0	0 ×	0
1	0	1
	1 ×	
2	0	2
	1 ×	
	2 ×	
3	0	3
	1 ×	
	2 ×	
	3 ×	
n		

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

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

⇒
```
p = 0;
for (i = 1; p <= n; i++)
{
    p = p + i;
}
```

i	p
1	0+1 = 1
2	1+2 = 3
3	1+2+3 = 6
4	1+2+3+4 = 10
⋮	
k	1+2+3+ ···· +k

When stop Any is $p > n$

$\therefore \quad p = \frac{k(k+1)}{2}$

$\frac{k(k+1)}{2} > n$

$k^2 > n$ $\quad O(\sqrt{n})$

$k > \sqrt{n}$

Algorithm

Iterative

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

≡

Recursive

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

(i) Any recursion can be converted into Iterative and any iterative can be converted into recursion.

(ii) If there is no algorithm exist either Iterative or recursive than it does not depends on the size 'n'. So time taken is constant.

Ex:-
```
A()
{
 i=1; s=1;
 while (s <= n)
 {
  i++
  s = s + i
  pf ("ravi");
 }
}
```

s	1	3	6	10	15	21	.....	$\frac{k(k+1)}{2}$ ← n
i	1	2	3	4	5	6	....	k iterations

's' is sum of natural numbers.

$\frac{k(k+1)}{2} > n$

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

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

Ex 1-

```
A()
{
  i = 1          i <= √n
for (i=1; i² <= n; i++)
   pf ("ravi");
}
```

$O(\sqrt{n})$

If there is any break statement inside the loop then we consider differently.

In this case, best case, Avg. case & worst case all has $O(\sqrt{n})$ complexity. Then, so assume theta notation also

Ex 2-

```
A()
{
  int i, j, k, n;
  for (i=1; i<=n; i++)
    {
      for (j=1; j<=i; j++)
        {
          for (k=1; k<=100; k++)
            {
              pf ("ravi");
            }
        }
    }
```

i = 1	i = 2	i = 3	-----	i = n
j = 1 time	j = 2 times	j = 3 times	-----	j = n times
k 100 times	k = 2*100	k = 3*100	-----	k = n*100

Total time $= 100 + 2*100 + 3*100 + \cdots + n*100$

$= 100(1 + 2 + 3 + \cdots + n)$

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

$= O(n^2)$

Ex:-

```
A()
{
  int i, j, k, n;
for (i=1; i<=n; i++)
{
 for (j=1; j<=i^2; j++)
 {
  for (k=1; k<=n/2; k++)
  {
   Pf ("ravi");
  }
 }
}
```

$i=1$	$i=2$	$i=3$	...	$i=n$
$j=1$ time	$j=4$ time	$j=9$ times	...	$j=n^2$
$k=n/2*1$	$k=n/2*4$	$k=\frac{n}{2}*9$	...	$k=\frac{n}{2}*n^2$

$$= \frac{n}{2}*1 + \frac{n}{2}*4 + \frac{n}{2}*9 + \dots + \frac{n}{2}*n^2$$

$$= \frac{n}{2}\left[1+4+9+\dots+n^2\right] \quad \text{[Sum of square of } n \text{ natural no.]}$$

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

$$= O(n^4)$$

$\because f(n) = n^k + n^{k-1} + \dots + 1 = O(n^k)$

Ex:-

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

$i = 1, 2, 4, \dots, n$

$2^0, 2^1, 2^2, \dots, 2^k$

$2^k = n$

$k = \log n \qquad O(\log_2 n)$

If `for (i=1; i<n; i=i*3)` than $O(\log_3 n)$

Ex:-

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

$n/2 * n/2 * \log_2 n = O(n^2 \log_2 n)$

Ex:-

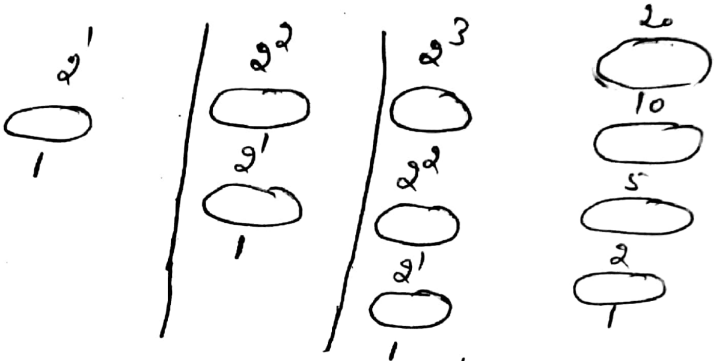
```
A()
{
  int i,j,k;
  for(i=n/2 ; i<=n ; i++)          — n/2
    for(j=1 ; j<=n ; j=2*j)        — log2n
      for(k=1 ; k<=n ; k=k*2)      — log2n
        pf(ravi);
}
```

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

$$= O(n(\log_2 n)^2)$$

Ex:- Assume $n \geq 2$

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



$\lfloor \log_2 20 \rfloor = \lfloor 4.4 \rfloor = 4$

$n = 2^k$ In this after k iterations

$k = \log_2 n$

$O(\lfloor \log_2 n \rfloor)$

⇒ If $n = n/5$ then answer will be $O(\lfloor \log_5 n \rfloor)$

Ex:-

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

i=1	i=2	i=3	i=k	i=n
j=1 to n	j=1 to n	j=1 to n	... j=1 to n	i=n/n
n times	1,3,5,... n/2 times	1,4,7,... n/3 times	n/k	n/n

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

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

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

Ex1-

```
A( )
{
  int n = 2^(2^k) ;
  for ( i=1 ; i<=n ; i++ )
   {
     j = 2
     while ( j<=n )
      {
        j = j^2
        Pf ("ravi");
      }
   }
}
```

k=1	k=2	k=3
n=4	n=16	$n = 2^{2^3} = 2^8$
j = 2, 4	j = 2, 4, 16	$j = 2, 2^2, 2^4, 2^8$
n * 2 times	n * 3 times	n * 4 times

$n * (k+1)$ times

where $n = 2^{2^k}$

$\log_2 n = 2^k$

$k = \log \log_2 n$

$O(n * (\log \log_2 n))$

Comparison of functions

1st Method :-

⇒

n	n^2	n^3
2	$2^2 = 4$	$2^3 = 8$
3	$3^2 = 9$	$3^3 = 27$
4	$4^2 = 16$	$4^3 = 64$

Yes, $n^2 < n^3$ This is one method to finding comparison of function by putting some val

⇒ 2nd Method :- n^2 n^3

Apply Log on both sides

$\log n^2$ $\log n^3$

$2 \log n < 3 \log n$

$\log ab = \log a + \log b$

$\log \frac{a}{b} = \log a - \log b$

$\log a^b = b \log a$

$a^{\log_c b} = b^{\log_c a}$

$a^b = n$ then $b = \log_a n$

Ex:- $f(n) = n^2 \log n$ $g(n) = n (\log n)^{10}$

Apply Log both sides

$\log (n^2 \log n)$ $\log (n (\log n)^{10})$

$\log n^2 + \log \log n$ $\log n + 10 \log \log n$

$2 \log n + \log \log n$ $\log n + 10 \log \log n$

Check this $2 \log n$ is bigger than $\log n$ than higher term $\log n$ is greater as $\log \log n$

So, $n^2 \log n > n (\log n)^{10}$

Ex:- $f(n) = 3n^{\sqrt{n}}$ $\quad g(n) = 2^{\sqrt{n}\log_2 n}$

$3n^{\sqrt{n}}$ $\quad 2^{\log_2(n^{\sqrt{n}})}$ $\quad [\because a^{\log_c b} = b^{\log_c a}]$

$3n^{\sqrt{n}}$ $\quad (n^{\sqrt{n}})\log_2 2$

$3n^{\sqrt{n}}$ $\quad n^{\sqrt{n}}$

$3n^{\sqrt{n}} > n^{\sqrt{n}}$

Asymptotically both are equal.

But value wise $3n^{\sqrt{n}} > 2^{\sqrt{n}\log_2 n}$

Ex:- $f(n) = n^{\log n}$ $\quad g(n) = 2^{\sqrt{n}}$

Apply log

$\log n^{\log n}$ $\quad \log_2 2^{\sqrt{n}}$

$\log n * \log n$ $\quad \sqrt{n}\log_2 2$

$\log^2 n$ $\quad \sqrt{n} = n^{1/2}$

If you are unable to judge still you can apply log :-

$2\log\log n$ $\quad \log n^{1/2}$

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

Ex:- $f(n) = 2^{\log n}$ $\quad g(n) = n^{\sqrt{n}}$

$\log n \log_2 2$ $\quad \sqrt{n}\log n$

$\log n < \sqrt{n}\log n$

Ex:- $f(n) = 2n$ $\quad g(n) = 3n$

Both these functions equal asymtotically.

Ex:- $f(n) = 2^n$ $\quad g(n) = 2^{2n}$

$\log 2^n$ $\quad \log 2^{2n}$

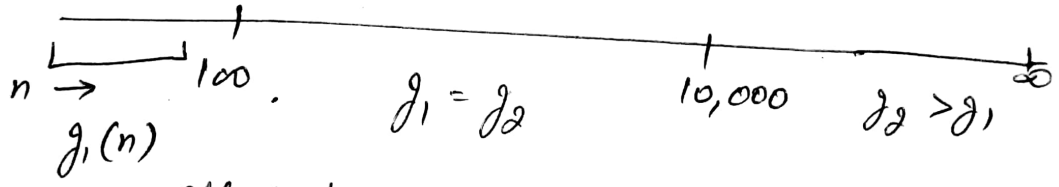
$n \log_2 2$ $\quad 2n \log_2 2$

n $\quad 2n$

Don't say equal because after log you just compare which is largest. You initially see that $f(n) < g(n)$ ~~asymptotically~~

Ex:- $g_1(n) = \begin{cases} n^3 & n < 100 \\ n^2 & n \geq 100 \end{cases}$

$g_2(n) = \begin{cases} n^2 & n < 10,000 \\ n^3 & n \geq 10,000 \end{cases}$



After 10,000 $g_2 > g_1$.

You don't take small value.

(1) True or False

$(n+k)^m = O(n^m)$ (True) $\quad (n+3)^2 = O(n^2)$

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

(3) $2^{2n} = O(2^n)$ (False)

(4) $\sqrt{\log n} = O(\log \log n)$ (False)

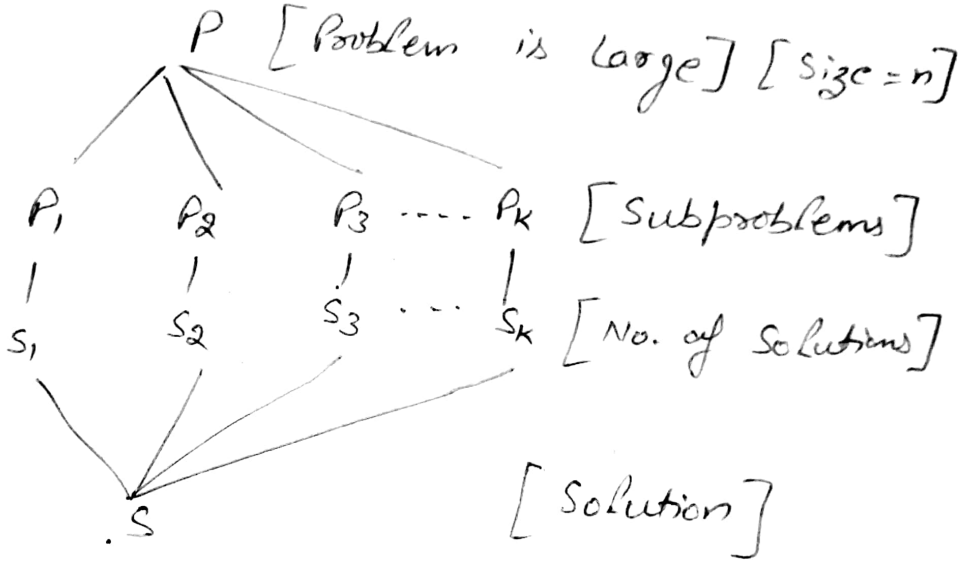
(5) $n^{\log n} = O(2^n)$ (True)

Divide and Conquer.

This topic is related to strategy solve a particular problem.

Stratgy is an approach or design for solving problem.

⇒ We can follow some guidlines to apply on particular problem which strategy is applicable. There is no formula.



⇒ If P problem is sort than P_1 subproblems also be sorting some elements and P_2 also do same thing. It means follow the same nature.

⇒ If this will happen, it can call problem recursively.

⇒ One important thing, if you broken the problems into some subproblems then one method we have to combine the solutions

⇒ For example, Workshop organised in college and problem P_1 has invitation, P_2 has Presentation and so on. Then it will not perform Divide & Conquer strategy.

⇒

```
DAC (P)
{
   if (small (P))
   {
      S(P);
   }
   else
   {
      divide P into P1, P2, P3 .... Pk
      Apply DAC (P1), DAC (P2), ....
      Combine ( DAC (P1), DAC (P2) .... )
   }
```

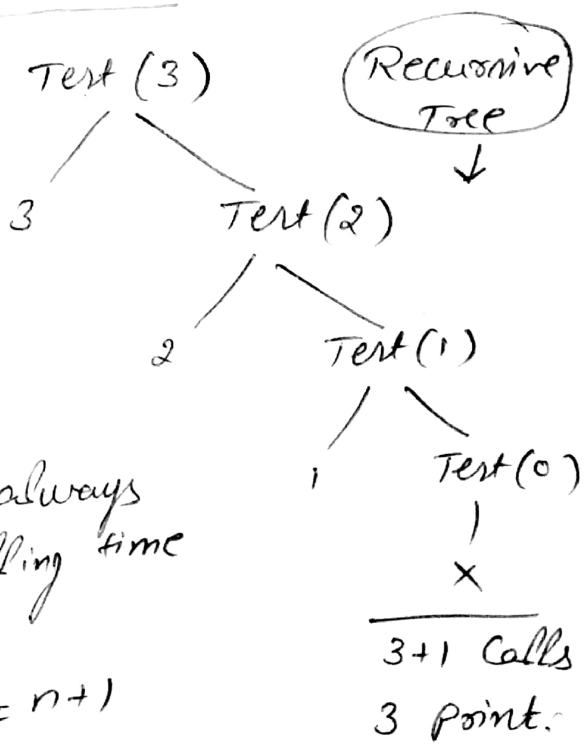
Recursive Property

Divide and Conquer Methods :—

(i) Binary Search
(ii) Merge Sort
(iii) finding Maximum and Minimum
(iv) Quick Sort
(v) Strassen's Matrix Multiplication

How to write Recurrence Relation :—

```
Void Test (int n)
{
   if (n>0)
   {
      Printf ("%d", n);
      Test (n-1);
   }
}
```



We can always take calling time i.e.

$f(n) = n+1$

So, $O(n)$

You can also prepare recurrence relation :—

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

```
T(n)— void Test (int n)
       {
         if (n>0)
         {
1 ——      Printf ("%d", n);
T(n-1) —— Test (n-1);
       }
       }
```

By using Substitution method :—

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

Substitute $T(n-1)$

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

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

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

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

⋮ continue for k times

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

Assume $n-k = 0$ [It is go substitution and reach 0]

$$n = k$$

$$T(n) = T(0) + n$$

$$T(n) = 1 + n \qquad O(n)$$

Ex:-

$$T(n) = \begin{cases} 1 & n=1 \\ T(n-1)+\frac{1}{n} & n>1 \end{cases}$$

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

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

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

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

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

$$= \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n}$$

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

Ex:-

$$T(n) = \begin{cases} 1 & n=1 \\ T(n-1) + \log n & \text{otherwise} \end{cases}$$

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

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

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

$$\vdots$$

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

$$= 1 + \log(n!) = 1 + \log(n^n) = 1 + n\log n$$

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

Ex:-

$$T(n) = \begin{cases} 1 & n=1 \\ 2T\left(\frac{n}{2}\right) + n & n>1 \end{cases}$$

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

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

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

$$= 2^2\left[2T\left(\frac{n}{8}\right) + \frac{n}{4}\right] + 2n$$

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

$$= \vdots$$

Assume $n = 2^k$

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

$$= 2^k T(1) + kn$$

$$= 2^{\log n} + n\log n$$

$$= n^{\log 2} + n\log n \Rightarrow n + n\log n$$

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

Ex:- $T(n) = \begin{cases} 1 & n=1 \\ 8T\left(\frac{n}{2}\right)+n^2 & n>1 \end{cases}$

$T(n) = 8T\left(\frac{n}{2}\right) + n^2$

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

$= 8\left[8T\left(\frac{n}{2^2}\right) + \frac{n^2}{4}\right] + n^2$

$= 8^2 T\left(\frac{n}{2^2}\right) + 2n^2 + n^2$

$= 8^2\left[8T\left(\frac{n}{2^3}\right) + \left(\frac{n}{2^2}\right)^2\right] + 3n^2$

$= 8^3 T\left(\frac{n}{2^3}\right) + 4n^2 + 3n^2$

$\vdots$

$2^k = n$

$= 8^k T\left(\frac{n}{2^k}\right) + n^2\left(2^0 + 2^1 + 2^2 + \cdots + 2^k\right)$

$= 8^{\log n} T(1) + n^2 \sum_{i=0}^{k} 2^i$

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

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

$= O(n^3)$

Master Theorem :-

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

$a \geq 1$, $b > 1$, $k \geq 0$ and p is real numbers.

1) If $a > b^k$, then $T(n) = \Theta(n^{\log_b a})$

2) If $a = b^k$

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

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

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

3) If $a < b^k$

   a) If $p \geq 0$, then $T(n) = \Theta(n^k \log^p n)$

   b) If $p < 0$, then $T(n) = O(n^k)$

$(\log n)^2$	, $\log^2 n$
$\log n \times \log n$	$\log \log n$
Both are different	

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

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

We should always compare a with b^k, we get

$a = 3$

$b^k = (2)^2 = 4$ then $3 < 4$

Case 3 will apply

3) a $\quad T(n) = \Theta(n^k \log^p n)$

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

$= \Theta(n^2)$

Problem 2 :- $T(n) = 4T(n/2) + n^2$

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

a	b^k
4	2^2

$4 = 4$

Apply (2) (a) :- $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$

$= \Theta(n^{\log_2 4} \log n)$

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

Problem (3):- $T(n) = T(n/2) + n^2$

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

a	b^k
1	2^2

$1 < 4$

Apply (3)(a):- $T(n) = O(n^2 \log^0 n)$

$= O(n^2)$

Problem (4):-

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

In this case a should be ($a \geq 1$) constant but this is not constant. So master theorem cannot be applied on this.

Problem (5):-

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

$a=16,\ b=4,\ k=1,\ p=0$

$16 > 4^1$

Case (1):- $T(n) = O(n^{\log_b a})$

$= O(n^{\log_4 16})$

$= O(n^2)$

Problem (6):-

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

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

$2 = 2^1$

Case (2)(a):- $T(n) = O(n^{\log_b a} \log^{p+1} n)$

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

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

Problem (7):-

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

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

$a=2,\ b=2,\ k=1,\ p=-1$

$2 = 2^1$

Case (2)(b):- $T(n) = O(n^{\log_b a} \log \log n)$

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

Problem (8):-

$T(n) = 2T\left(\frac{n}{4}\right) + n^{0.51}$

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

$2 < 4^{0.51}$

Case (3)(a):- $T(n) = O(n^k \log^p n)$

$= O(n^{0.51})$

Problem (9):- $T(n) = 0.5\, T(n/2) + \frac{1}{n}$

$a = 0.5$

In our theorem, $a \geq 1$ but this is $a = 0.5$. So master theorem cannot be applied.

So, Not according to Master Theorem

Problem (10):- $T(n) = 6T\left(\frac{n}{3}\right) + n^2 \log n$

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

$6 < 3^2$

Case (3)(a):- $T(n) = O(n^k \log^p n)$

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

Problem (11):- $T(n) = 64\, T\left(\frac{n}{8}\right) - n^2 \log n$

Whenever you have (-) minus symbol. That is not according to master theorem question.

Problem (12):- $T(n) = 7T(n/3) + n^2$

$a = 7, b = 3, k = 2, p = 0$

$7 < 3^2$

Case (3)(a) $T(n) = O(n^k \log^p n)$

$= O(n^2 \log^0 n)$

$= O(n^2)$

Problem (13):- $T(n) = 4T(n/2) + \log n$ (13)

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

$4 > 2^0$

Case (1):- $T(n) = O(n^{\log_b a})$

$= O(n^{\log_2 4})$

$= O(n^2)$

Problem (14):- $T(n) = \sqrt{2}\, T(\frac{n}{2}) + \log n$

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

$\sqrt{2} > 2^1$

Case (1):- $T(n) = O(n^{\log_b a})$

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

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

$= O(\sqrt{n})$

Problem (15):- $T(n) = 2T(\frac{n}{2}) + \sqrt{n}$

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

$2 > 2^{1/2}$

Case (1):- $T(n) = O(n^{\log_b a})$

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

$= O(n)$

Problem (16):- $T(n) = 3T(\frac{n}{2}) + n$

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

$3 > 2^1$

Case (1):- $T(n) = O(n^{\log_b a})$

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

Problem (17):- $T(n) = 3T(\frac{n}{3}) + \sqrt{n}$

$a = 3, b = 3, k = \frac{1}{2}, p = 0$

$3 > 3^{1/2}$

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

$= O(n^{\log_3 3})$

$= O(n)$

Problem (18):- $T(n) = 4T(\frac{n}{2}) + cn^1$

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

$4 > 2^1$

Case (1):- $T(n) = O(n^{\log_b a})$

$= O(n^2)$

Problem (19):- $T(n) = 3T(\frac{n}{4}) + n \log n$

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

$3 < 4^1$

Case (3)(a):- $T(n) = O(n^k \log^p n)$

$= O(n \log n)$

Merge Sort

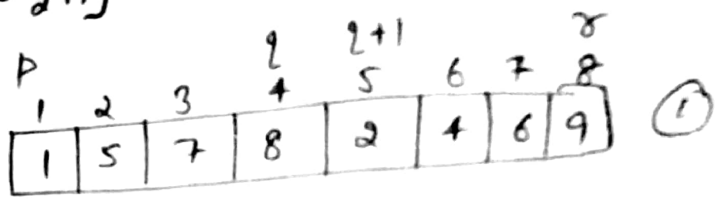
It is better than Insertion sort. Sometimes it is better Quick sort in terms of space complexity.

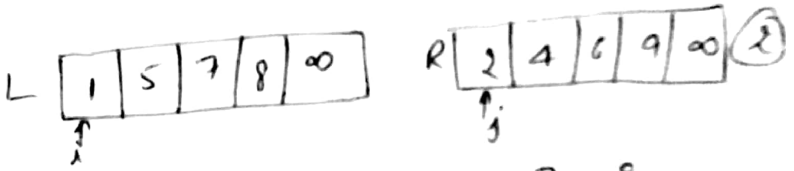
Heart of the Merge sort :-

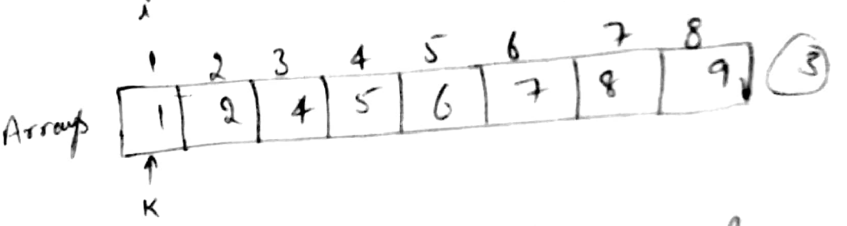
Merging :-(Out of place procedure) → Three pointers

```
MERGE (A, p, q, r)
{
  n1 = q - p + 1
  n2 = r - q          } No. of elements in 1st List & 2nd List.
  Let L[1 ... n1+1] and R[1 to n2+1]
  be new arrays
  for (i = 1 to n1)                  } Copy the value in Left array
    L[i] = A[p+i-1]
  for (j = 1 to n2)                  } Copy the value in right array
    R[j] = A[q+j]
  L[n1+1] = ∞
  R[n2+1] = ∞
  i = 1, j = 1
  for (k = p to r)
    if (L[i] ≤ R[j])
      A[k] = L[i]
      i = i+1
    else A[k] = R[j]
      j = j+1
```

Time Complexity O(n)
Space Complexity O(n)







→ ∞ means very large value.

→ From (1) to (2) every element copy into Left & Right than time is O(n).

→ From (2) to (3), there are n comparisons & n value is copied. so total time taken is O(n+n) = O(n)

For example:-

'n' elements in 1st sorted List

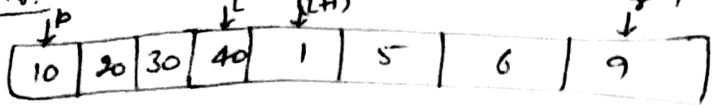
m elements in 2nd sorted List

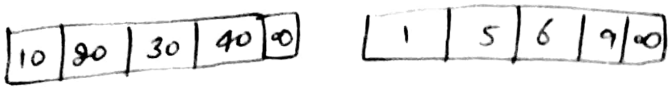
(n+m) total size of the result array

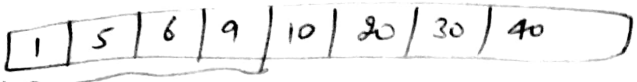
O(n+m) time complexity.

Q:- Why we use infinity in the end ?

Ans:- Take some another example :-







You seen that one entire list is copied here and other list is still there. So we have to second list is copied into array than compare with ∞.

If ∞ is not there than we change the code to copy the other list.

→ Merge - Sort (A, p, r)

{

if p < r

q = ⌊(p+r)/2⌋

merge - sort (A, p, q)

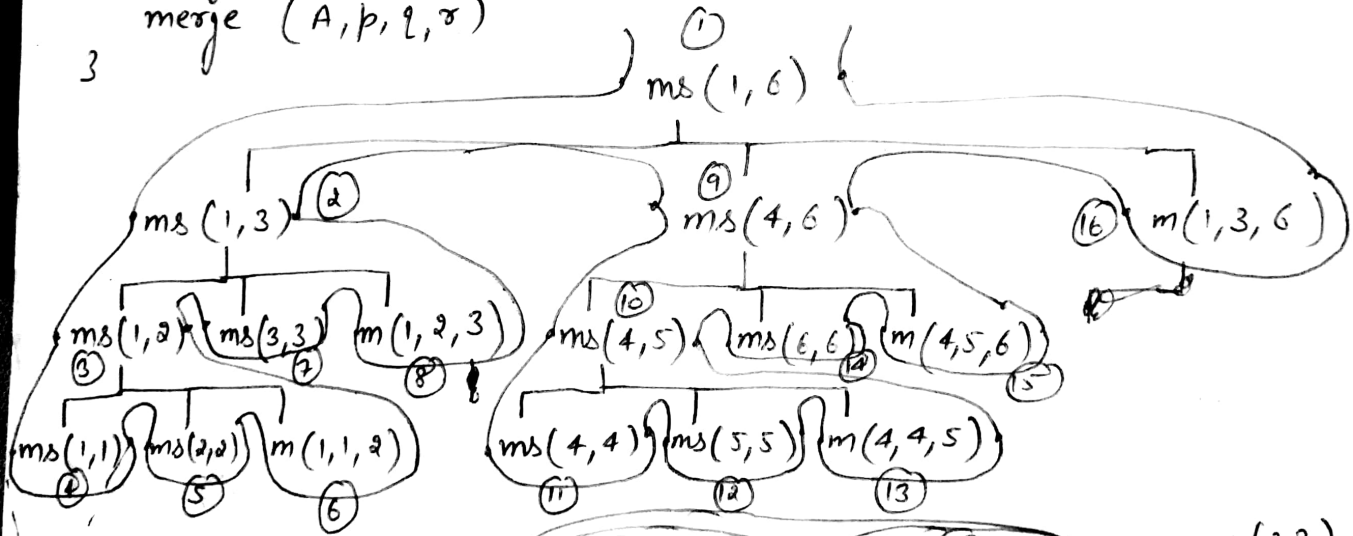
merge - sort (A, q+1, r)

merge (A, p, q, r)

}

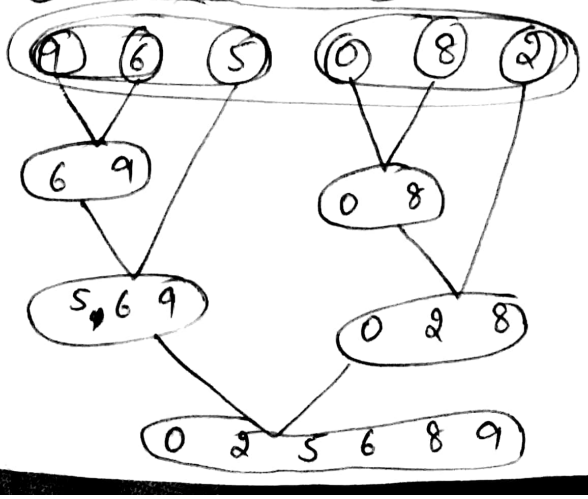
⇒ Divide the array into such a small problem which contains one ~~merg~~ element

⇒ Combine the list by using merge procedure.



It means it done

⇒ Every ~~Array~~ List which containing one element it means sorted.



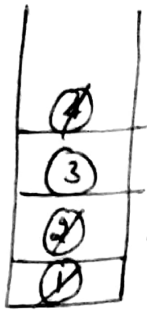
⇒ ms(3,3) you applying sorted list

⇒ m(1,2,3) it means merging (1,2) and (3,3)

⇒ Q:- Do we really need require height of 16 stacks ?

Ans:- No, it actually mean height of stack is equal to the height of the tree.

For ex:-



5 5 6 11 12 13
7 8 10 14 15
9 16

5 is push on the same cell privously 4 placed.

→ At level 1, only one function call.
→ At level 2, in tree three function call see same in the stack.

Therefore, total space require :—

elements (n) than $(\lceil \log n \rceil + 1)$ Levels

Size of stack $(\lceil \log n \rceil + 1)\, k$

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

k is constant amount of size a cell

For merging space require :- $O(n)$

Stack :- $O(\log n)$

Total space :- $O(n + \log n)$

Complexity $O(n)$

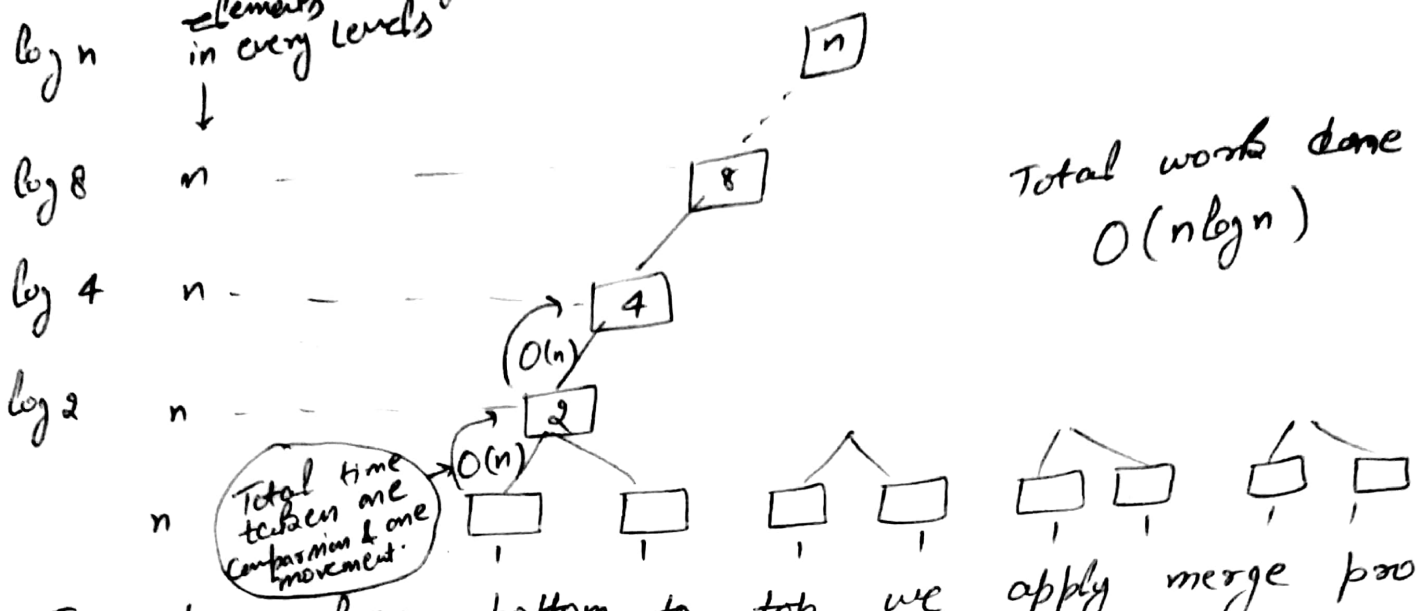
Time Complexity :-

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

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

If we have sorted & unsorted list it always give $(n \log n)$

Problem!- Given n elements, merge them into one sorted list using merge procedure.



Total work done is $O(n \log n)$

⇒ In above from bottom to top, we apply merge procedure to combine two sorted list

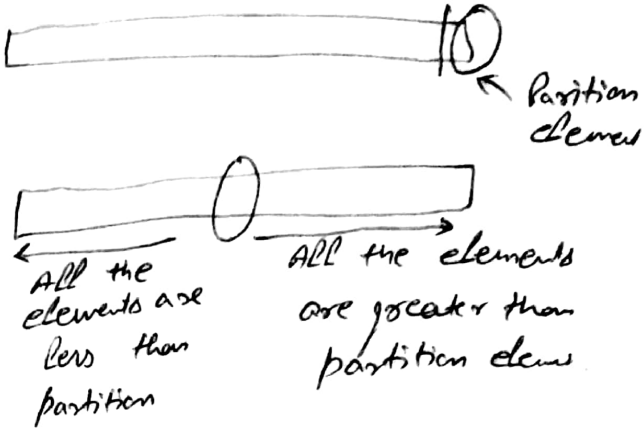
⇒ To reach size 1 to size 2 than cross no. of levels is 1 it means ($\log 2$)

⇒ To reach size 1 to size 4 than cross no. of levels is 2 ($\log 4$) & so on.

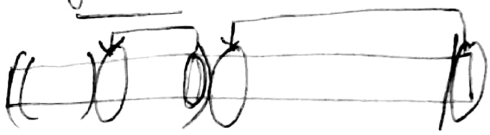
Quick Sort:- If the problem is small then quick sort is fastest and Merge sort not perform fastest because it has more function calling.

⇒ Heart of sorting is PARTITION.

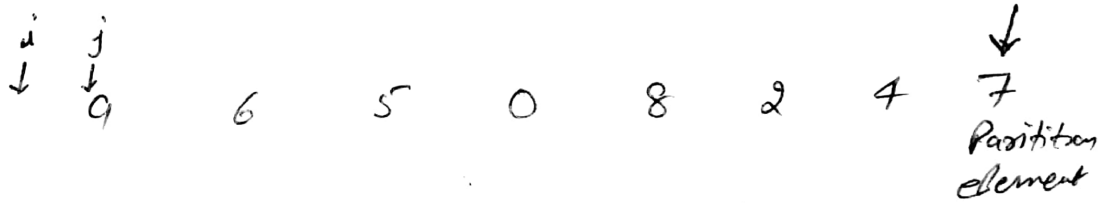
```
PARTITION (A, p, r)
{
   x = A[r]
   i = p - 1
   for (j = p to r-1)
   {
      if (A[j] ≤ x)
      {
         i = i + 1
         exchange A[i] with A[j]
      }
   }
   exchange A[i+1] with A[r]
   return i+1
}
```



⇒ After doing above step, than one element is fixed.

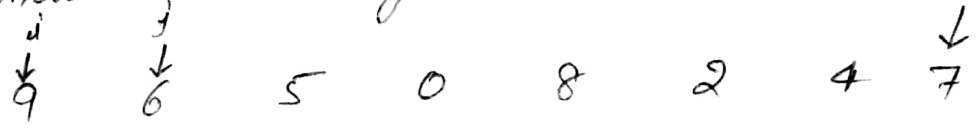


Example:-

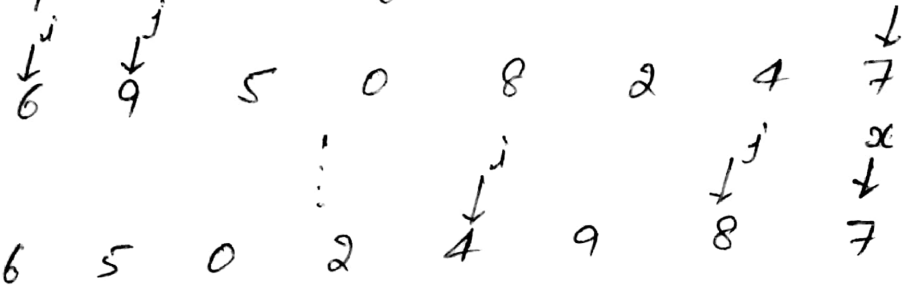


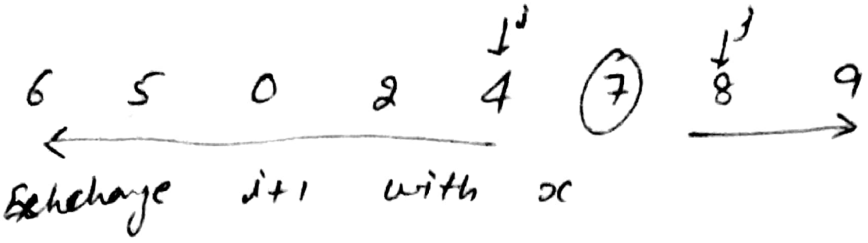
9 is greater than move j

Whenever j value is less than partition element than increment the i by 1.



Swap i and j.





Time taken by partitioning is $\boxed{O(n)}$

Q:- How partitioning is going to help us to build a Quick sort ?

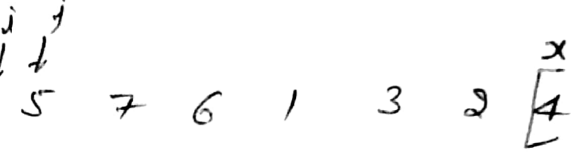
Ans:-

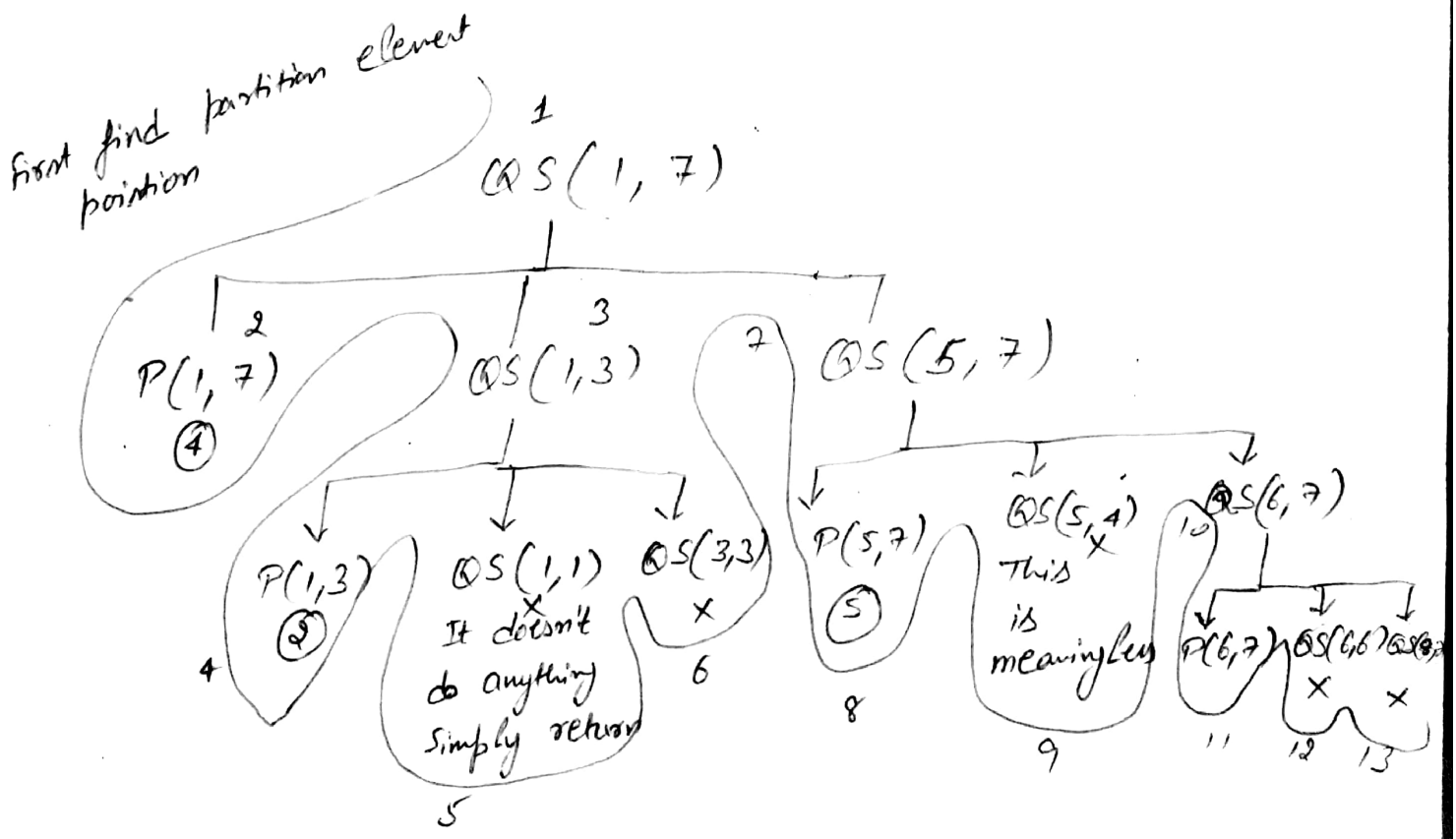
```
QUICKSORT (A, p, r)
{
  if (p < r)
  {
    q = PARTITION (A, p, r)
    QUICKSORT (A, p, q-1)
    QUICKSORT (A, q+1, r)
  }
}
```

(Pivot) Partition element:-

[First element / Last element
Any element]

Ex:-



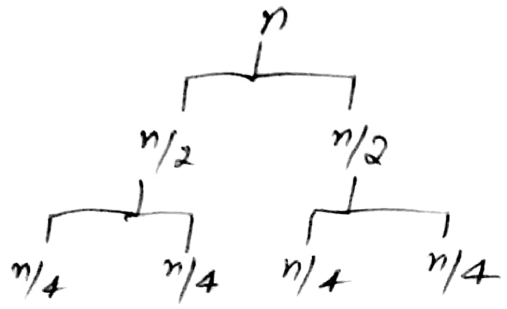


It requires 13 function calls but does not mean that require stack entries.

It means number of levels.

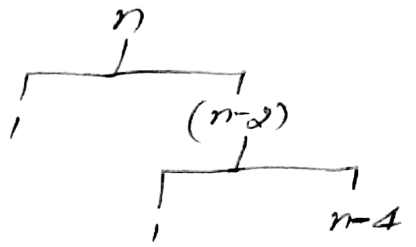
⇒ Space Complexity:- (If it is balanced)

Best Case:-



Q:- How many records require?
Ans:- No. of levels require records.
Assuming (log n) levels :-
$O(\log n)$.

Worst Case:-



[Balance is not given means not divide equal size]

Height of the tree might goes O(n)

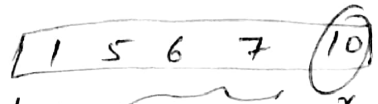
⇒ Time Complexity:-

	Best Case	Worst Case
QUICKSORT (A, p, r) ----	T(n)	T(n)
{		
if (p < r)		
{		
q = PARTITION (A, p, r) ---	O(n)	O(n)
QUICKSORT (A, p, q-1) ---	T(n/2)	T(0)
QUICKSORT (A, q+1, r) ---	T(n/2)	T(n-1)

Best Case:-

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

Worst Case:-



It already in fixed position

$T(n) = T(n-1) + O(n)$
$= T(n-1) + cn$
$= T(n-2) + c.(n-1) + c.n$
$= T(n-k) + c.(n-k-1) + \dots + c(n-1) + cn$
$= c.1 + c.2 + c.3 + \dots + c.n$
$= O(n^2)$

Introduction to Heaps

Depends on the I/P, complexity runs different.

	Insert	search	Find Min	Delete Min
Unsorted Array	$O(1)$	$O(n)$	$O(n)$	$O(n)$
(Non-dec) Sorted Array	$O(\log n)$	$O(\log n)$	$O(1)$	$O(n)$
Unsorted Linked List	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Min Heap	$O(\log n)$		$O(1)$	$O(\log n)$

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

↓ Time taken for mid element

↓ Either first half or second half.

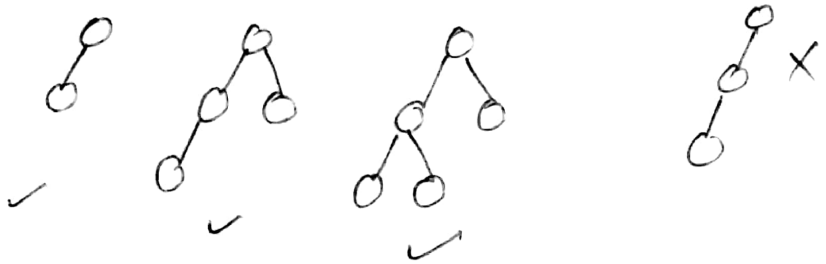
⇒ There are various commonly inputs such as unsorted array, sorted array & so on.

⇒ Our aim is insert, Find Min & Delete min should be optimal

Ans:- But here atleast one is $O(n)$. So we need heap.

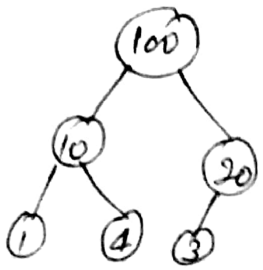
HEAP:- (i) Heap is a binary tree or 3-ary ..... n-ary tree.

(ii) Heap is an almost compete. It need not be complete tree.

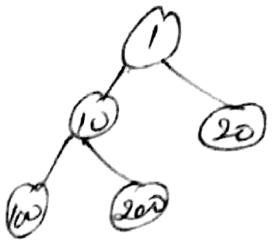


It means always start from left to Right in the last level.

Max Heap:- The root value should be maximum.

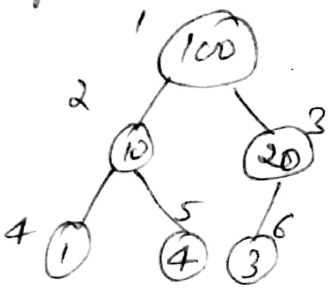


Min Heap:- Root contains the minimum value of the entire tree.



How it is implemented in array!—

Always take Max Heap here!—



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

If you find the parent index!—

$\lfloor i/2 \rfloor$ Ex:- $\lfloor 5/2 \rfloor = 2$

Find out the tree (Max-Heap) given corresponding array has follow conc

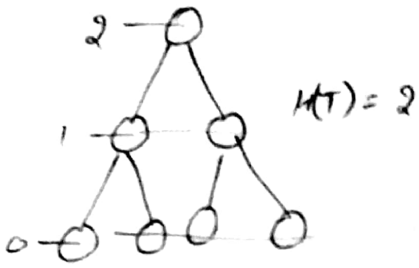
① Q:- Is given array is Max-Heap or not ?
② Q:- If it is not Max-Heap till what element it is Heap ?

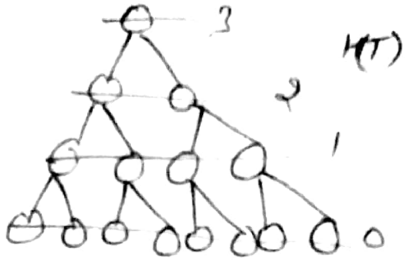
	A.length (Total elements in array)	A.HeapSize (How many elements follow Heap property)
(i) 25, 12, 16, 13, 10, 8, 14	7	7
(ii) 25, 14, 16, 13, 10, 8, 12	7	1
(iii) 25, 14, 13, 16, 10, 8, 12	7	2
(iv) 25, 14, 12, 13, 10, 8, 16	7	1
(v) 14, 13, 12, 10, 8	5	5
(vi) 14, 12, 13, 8, 10	5	5
(vii) 14, 13, 8, 12, 10	5	5
(viii) 14, 13, 12, 8, 10	5	5
(ix) 89, 19, 40, 17, 12, 10, 2, 5, 7, 11, 6, 9, 70	13	2

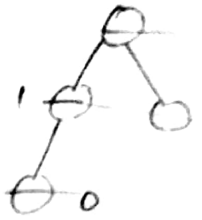
Before going to Algorithms, first know the properties :-

Height of Node :- No. of edges from that node to longest leaf nodes (or) longest possible edges from node to leaf nodes

H(T) = 1







[Almost Complete B.T]

It means not always the case every same level has same height

Height of Tree :- Height of root tree is height of tree.

Property :-

(i) Given the height h then find out the number of nodes :-

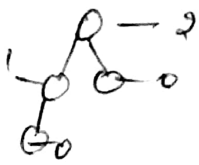
H	1	2	3	4
Max	3	7	15	

[Property apply to C.B.T not almost C.B.T]

Max. No. of Nodes = $2^{h+1} - 1$

(ii) If No. of nodes (n) is given in Almost C.B.T or C.B.T then height of the tree.

	No. of Nodes (n)	Height of tree $\lfloor \log n \rfloor$
C.B.T	3	$\lfloor \log_2 3 \rfloor = 1$
	7	$\lfloor \log_2 7 \rfloor = 2$
	15	$\lfloor \log_2 15 \rfloor = 3$

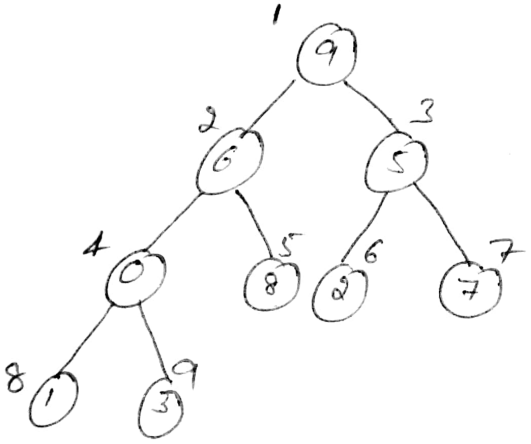


Almost C.B.T.	4	$\lfloor \log_2 4 \rfloor = 2$

Algo:- MAX_HEAPIFY (A, i)

```
{
   l = 2i;
   r = 2i + 1;
   if ( l ≤ A.heap_size and A[l] > A[i] )
        largest = l;
   else largest = i;
   if ( r ≤ A.heap_size and A[r] > A[largest])
        largest = r;
   if (largest ≠ i)
      exchange A[i] with A[largest]
      MAX_HEAPIFY (A, largest)
}
```

Ex:- 9 6 5 0 8 2 7 1



Leaf Nodes:-

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

$\lfloor 9/2 \rfloor + 1$ to 9

5 to 9

Non-Leaf Nodes:-

1 to $\lfloor \frac{n}{2} \rfloor$

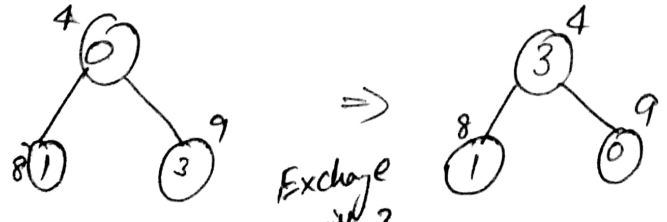
1 to $\lfloor \frac{9}{2} \rfloor$

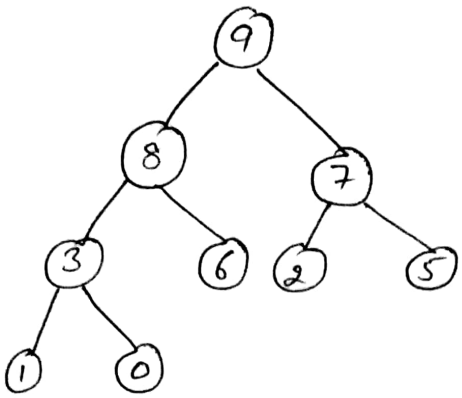
1 to 4

See which is the largest index for Non-Leaf:-

From 4 to 1.

First checks index 4.





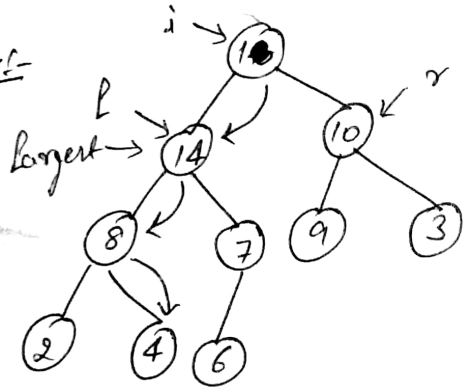
In this build the Max-Heap using MAX-HEAPIFY.

Points:-

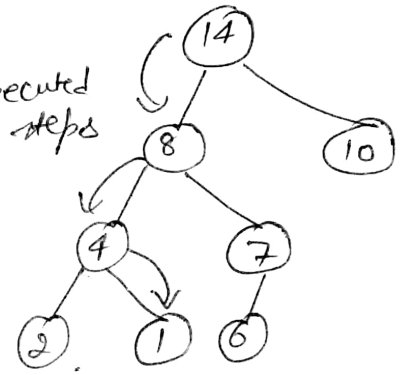
(i) Every leaf is a Max-Heap.

(ii) Find out the Largest index which is not a leaf I tried to apply HEAPIFY.

Ex:-



After executed three steps ⇒



This above Case happen if left subtree is a Max-Heap and right subtree also Max-Heap but root is not Max-Heap.

Q:- What is the distance & Comparisions?

Ans:- 2 Comparisions in each level

$(2 \times \log n)$

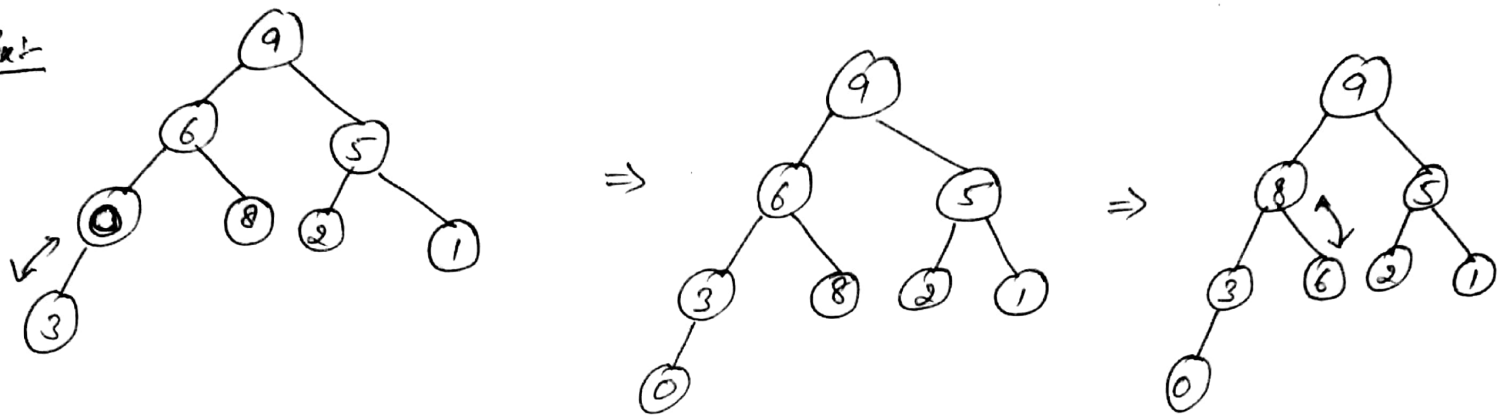
$O(\log n)$

Height of Heap always $O(\log n)$

Algo:- **BUILD MAX HEAP:-**

```
BUILD - MAX - HEAP (A)
{
  A.heap-size = A.Length
  for ( i = ⌊A.Length/2⌋ down to 1)
      MAX - HEAPIFY (A, i)
}
```

Ex:-



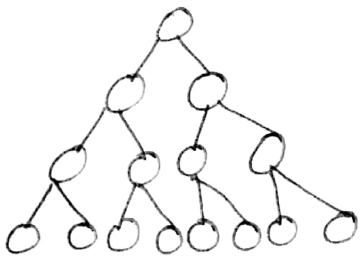
Q:- Time Complexity for build Max-Heap ?

Ans:- Time taken is $O(\log n)$ for Max-Heapify.

To build Heap accur nearly n time. So,

$$O(n \log n)$$

This is not the way. So, we move to next method!



(1, 3)
(2, 2)
(4, 1)
(8, 0)

No. of nodes ↑ Height ↑

h is given then max. no. of nodes will be $\left\lceil \frac{n}{2^{h+1}} \right\rceil$

$\left\lceil \frac{15}{2^{0+1}} \right\rceil = 8$, $\left\lceil \frac{15}{2^{1+1}} \right\rceil = 4$

$\left\lceil \frac{15}{2^{2+1}} \right\rceil = \left\lceil \frac{15}{8} \right\rceil = 2$ $\left\lceil \frac{15}{16} \right\rceil = 1$

Insertion Sort:-

```
Insertion_sort(A)
{
  for j = 2 to A.length
      key = A[j]
  // insert A[j] into sorted
     sequence A[1....j-1]
     i = j-1
  while (i>0 and A[i]>key)
      A[i+1] = A[i]
       i = i-1
  A[i+1] = key
}
```

For ex:-

Playing Card

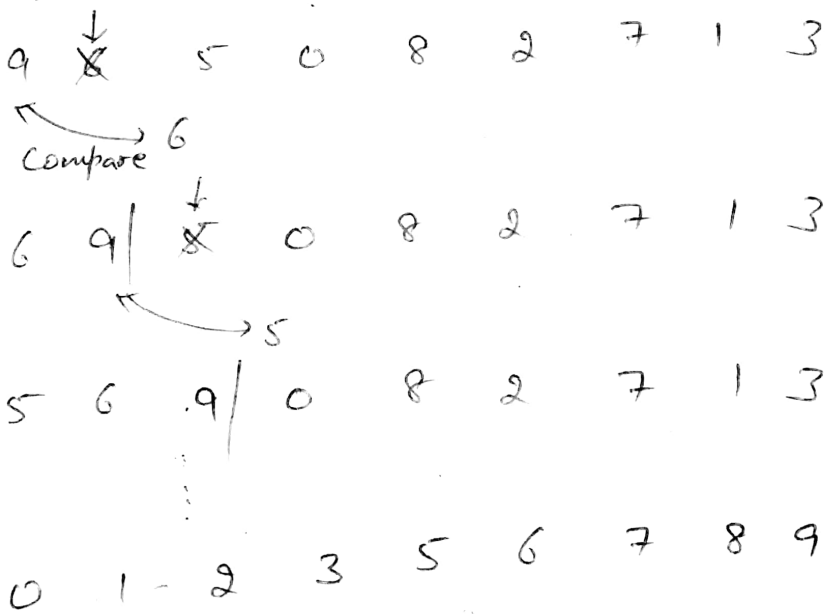
→ In this first picks the card & hold in Left Hand

→ Picks second card and compare with first card. If second card is greater put it in the right side.

→ Picks third card & compare from right to Left & place in position.

For ex:- 9 | 6 5 0 8 2 7 1 3

→ Picked first number and this is already sorted.



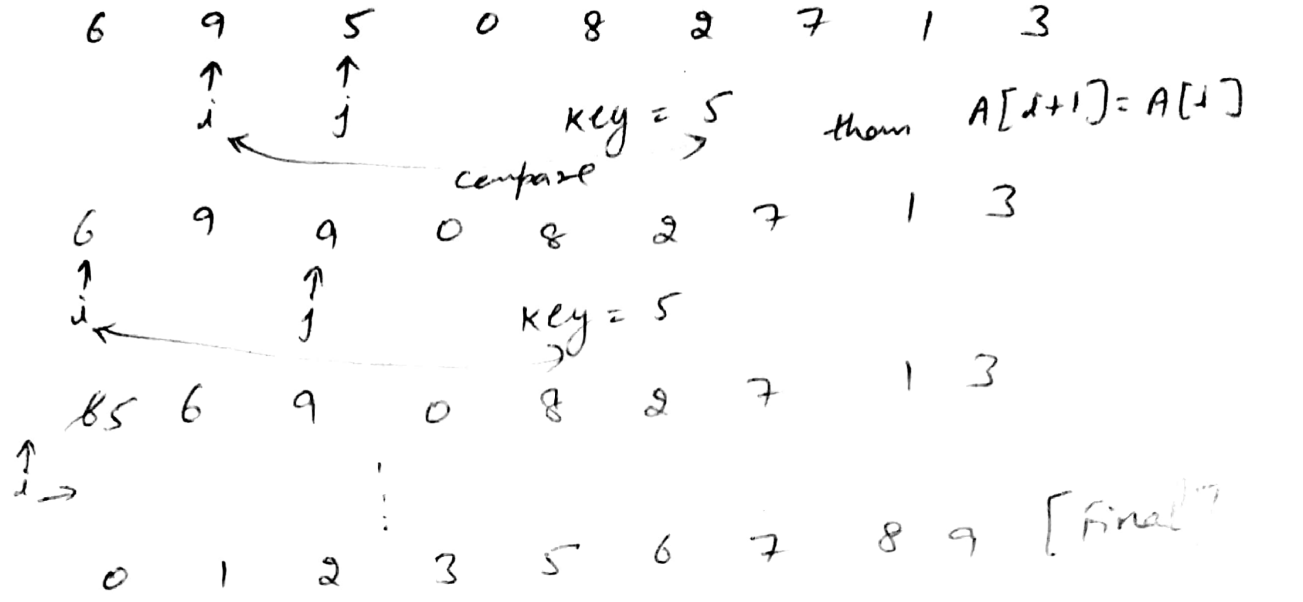
Apply Algorithm on above ex:-

(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)
9	6	5	0	8	2	7	1	3
↑ i	↑ j							

key = 6 [Tempory hold the current No.]

9 6	9	5	0	8	2	7	1	3

↑ i → i+1 Fit the key value after i = i+1



Worst Case :-

Comparsion + movements

If $j=2$ than $1 + 1 = 2 = 2(1)$

$j=3$ than $2 + 2 = 4 = 2(2)$

$j=4$ than $3 + 3 = 6 = 2(3)$

$\vdots$

$j=n$ than $(n-1) + (n-1) = 2(n-1)$

$2(1) + 2(2) + 2(3) + \dots + 2(n-1)$

$\frac{2(n-1)(n)}{2} = O(n^2)$ [It will happen when input is reversed sorted order]

Best Case :- Already sorted List

1 2 3 4 5 6 7

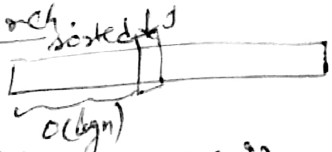
	$j=2$	3	4	5	$\dots$	$(n-1)$
Comparsions →	1	1	1	1		1

$O(n-1) = \Omega(n)$

we are not taking any extra space for storing then algorithm is known as Inplace.

Some points :-

⇒ If we want to decrease the comparsions then use Binary search instead of Linear Search



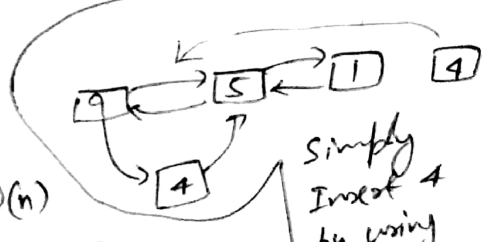
⇒

Comparsions	Movements
$O(\log n)$	$O(n) \rightarrow O(n)(n-1)$ times $\Rightarrow O(n^2)$

⇒ If we want to decrease the movements than use

⇒ Doubly Linked list

Comparsions	Movements
$O(n)$	$O(1) \Rightarrow O(n)$



Complexity can not be improved either by using Binary Search or Double Linked List

⇒ **(2,4) Trees :-**

- They are search trees (but not binary search trees).
- They are also known as 2-4, 2-3-4 trees.

Multi-way Search Trees :-

⇒ Each internal node of a multi-way search tree T:

(i) It has at least two children.

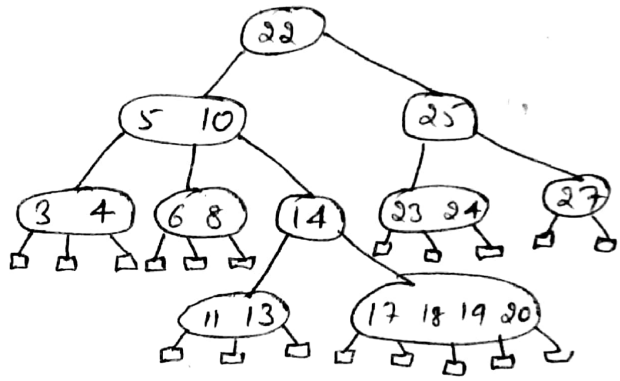
(ii) Stores a collection of items of the form (K, x), where K is a key and x is an element.

(iii) contains $d-1$ items, where d is the number of children

(iv) Has pointers to d children.

⇒ Children of each internal node are "between" items.

⇒ All keys in the subtree rooted at the child fall between keys of those items.

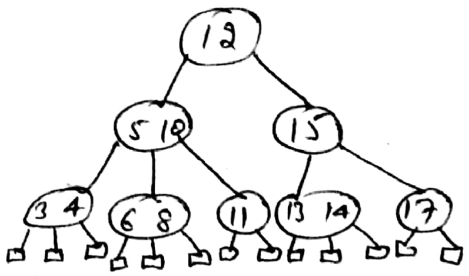


Multi-way Searching :-

- Similar to binary searching
  (i) If search key $s < K_1$, search the leftmost child.
  (ii) If $s > K_{d-1}$, search the rightmost child.
- What about an in-order traversal looks like?

3 4 5 6 8 10 11 13 14 17 18 19 20 22

Properties of (2,4) Trees :-



1) At most 4 children.
2) All leaf nodes are at the same level.
3) Height h of (2,4) has tree is at least $\log_4 n$ [Every node has 4 children] and at most $\log_2 n$ [Every node has 2 children].

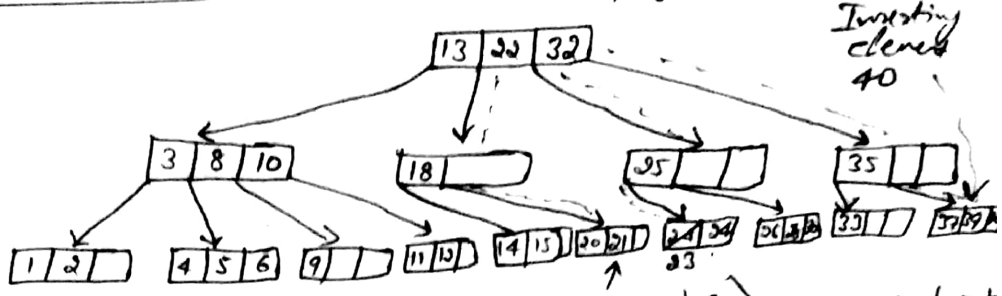
In this Every node require more than one comparsion because each node contains at most 3 keys but in BST require only one comparsion for each node.

Insertion in (2,4) trees :-

Just insert like BST.

Case 1:- No problem if the node has empty space.

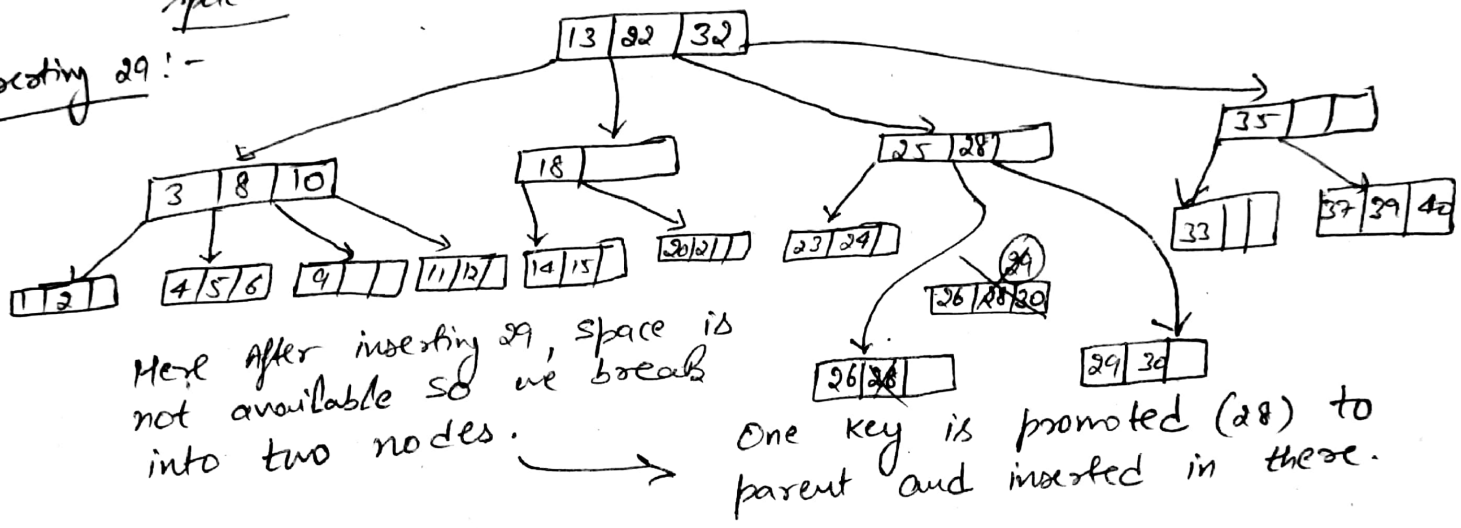
Inserting element 21 :-



Inserting element 23 :-

Case 2:- Nodes get split if there is insufficent space.

Inserting 29 :-

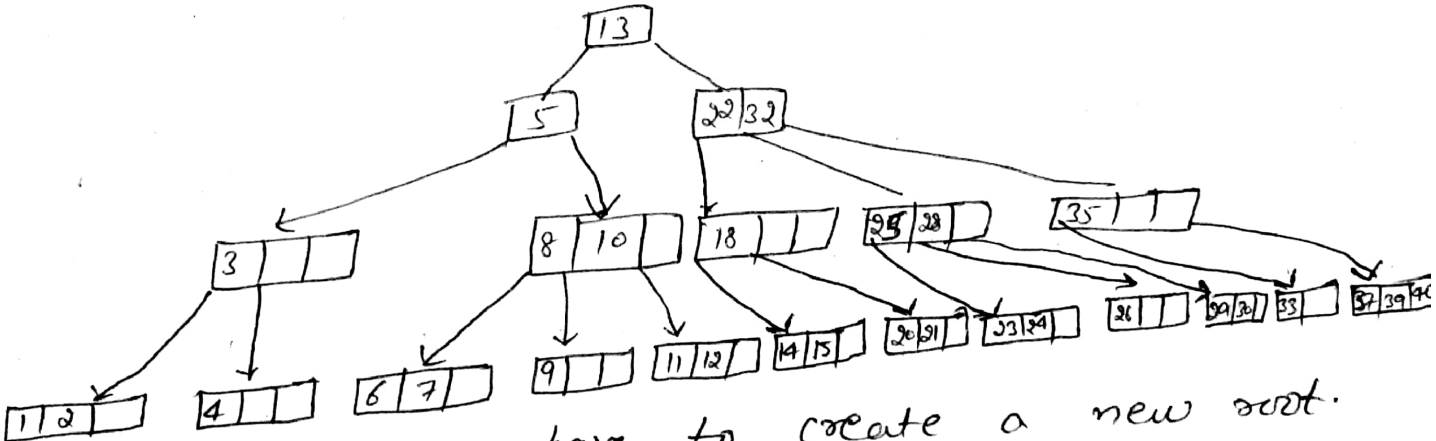


Here after inserting 29, space is not available so we breaks into two nodes. → One key is promoted (28) to parent and inserted in there.

Same apply for inserting 7 :-

Case 3:-
- If parent nodes does not have sufficent space then it is split.
- In this manner splits can cascade.

" larger key of smaller node ↑ promoted "



⇒ Eventually we may have to create a new root.

⇒ This increases the height of the tree.

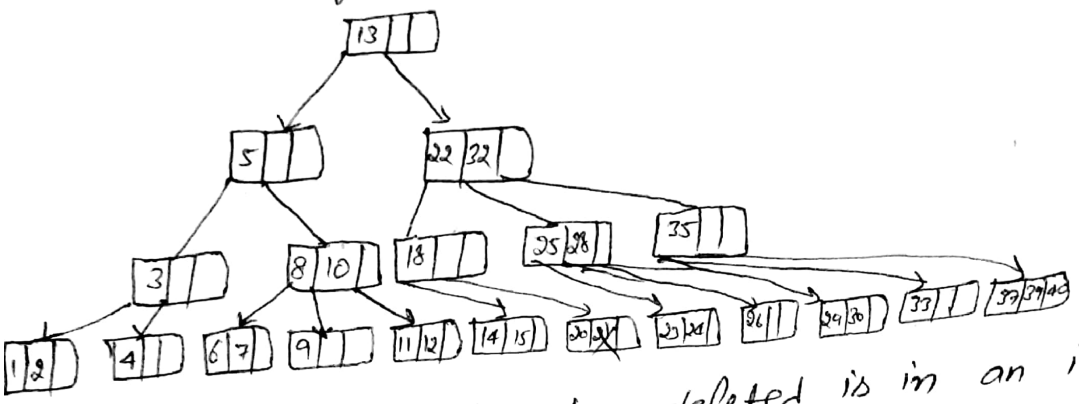
Time for Search & Insertion :-

- A search visits $O(\log N)$ nodes
- An insertion requires $O(\log N)$ node splits
- Each node split takes constant time.
- Hence, operations search and Insert each take time $O(\log N)$.

Deletion in (2,4) trees :-

Case 1 :- Delete 21

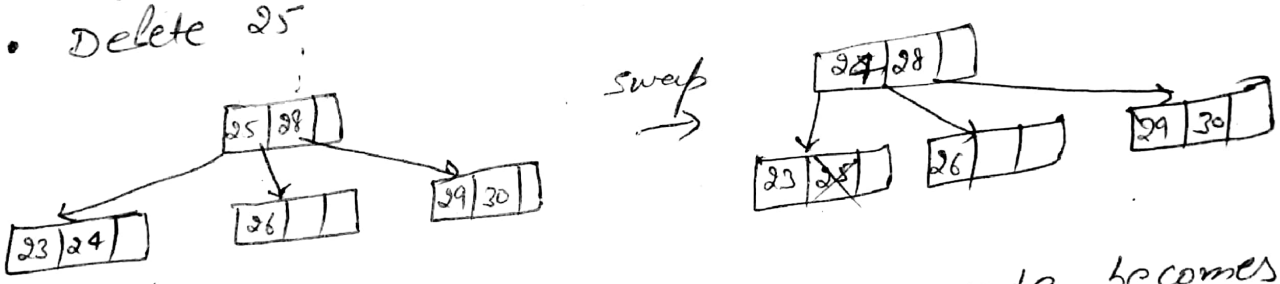
- No problem if key to be deleted is in a leaf with at least 2 keys.



Case 2 :- If key to be deleted is in an internal node then we swap it with its predecessor (which is in a leaf) and then delete it.

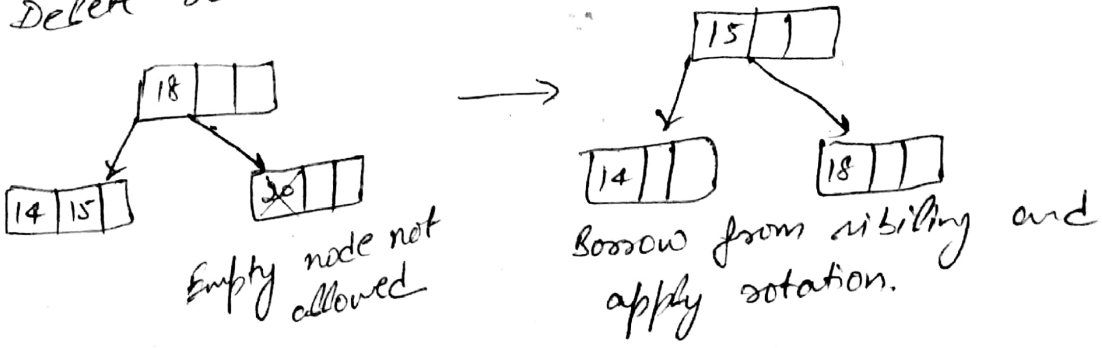
First Go left and keep going right.

- Delete 25



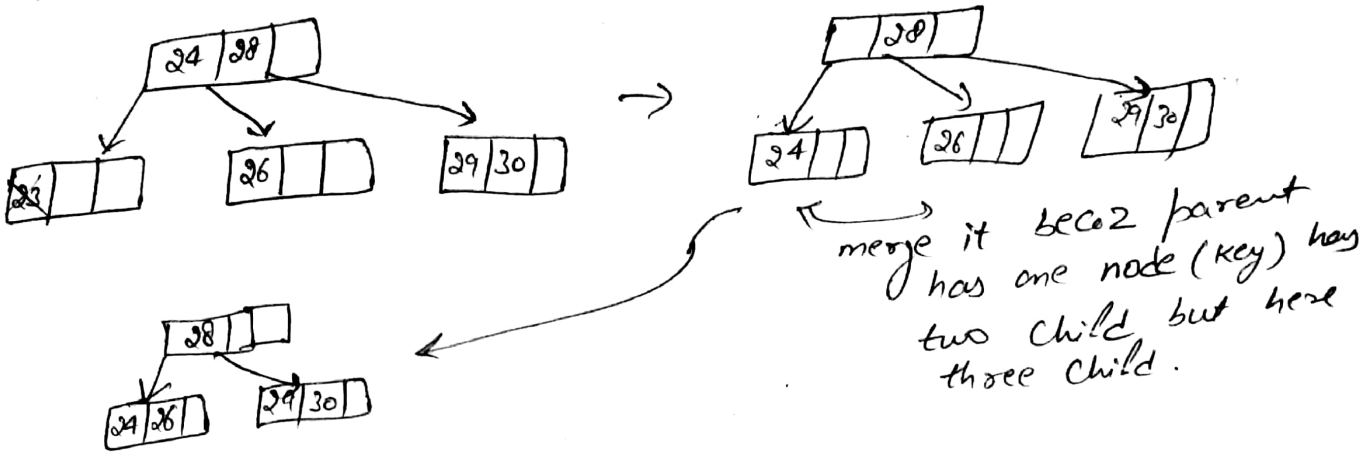
Case 3 :- • If after deleting a key a node becomes empty then we borrow a key from its sibling.

- Delete 20



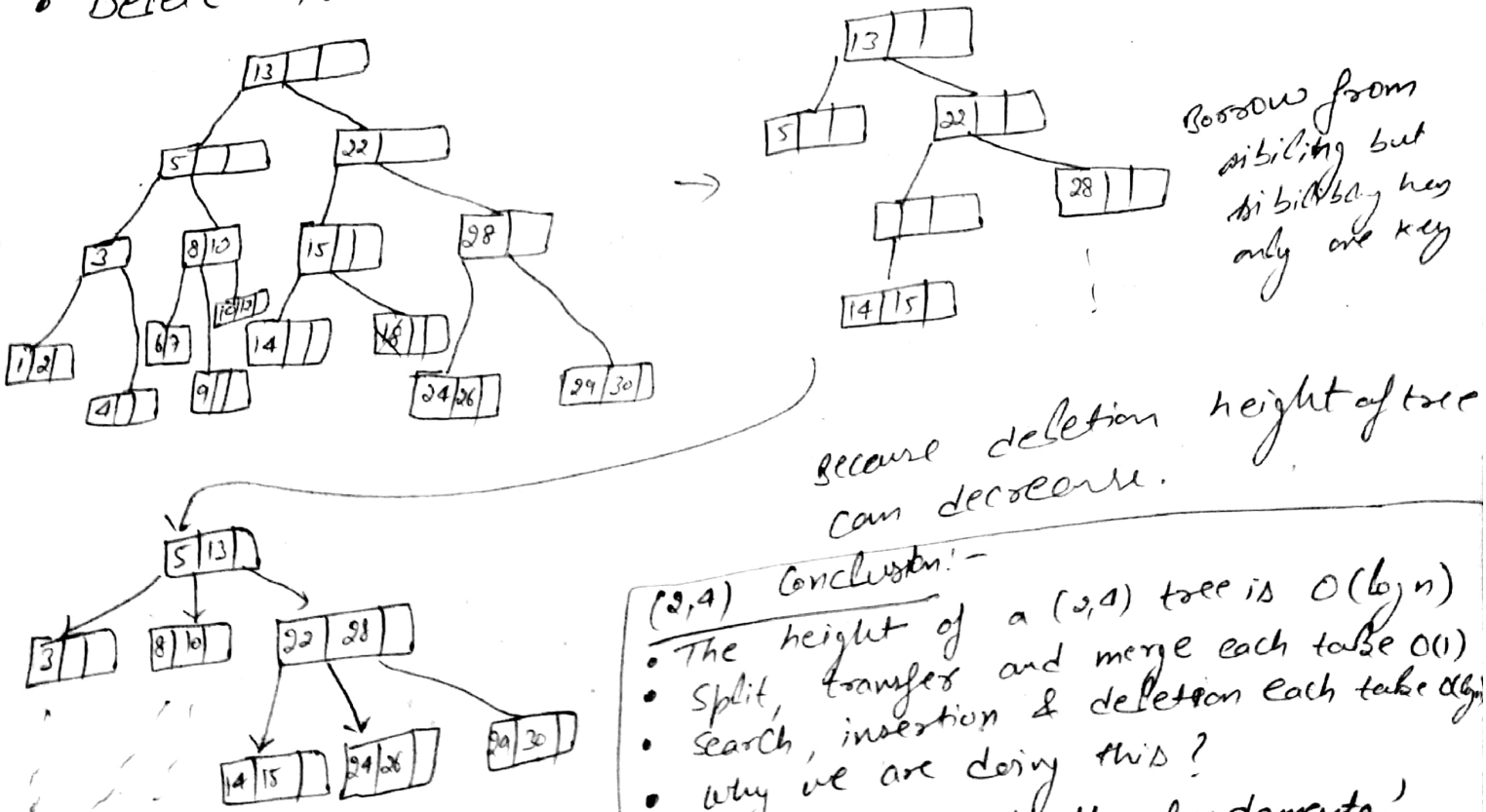
Case 4:-

- If sibling has only one key then we merge with it.
- The key in the parent node separating these two sibilings moves down into the merged node.
- Delete 23.



Case 5:-

- Moving a key down from the parent corresponds to deletion in the parent node.
- The procedure is the same as for a leaf node.
- Can lead to a cascade.
- Delete 18.



(2,4) Conclusion:-

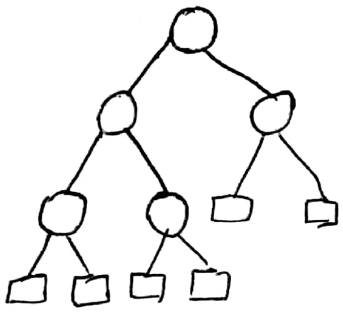
- The height of a (2,4) tree is $O(\log n)$
- Split, transfer and merge each take $O(1)$
- Search, insertion & deletion each take $O(\log n)$
- Why we are doing this?

Ans:- They are pretty fundamental to the idea of Red-Black trees as well.

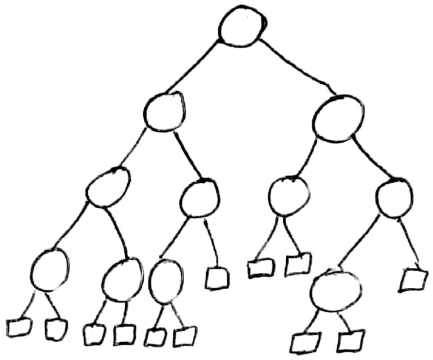
Red Black Trees

1) A red black tree is a binary search Tree in which each node is colored red or black.
2) The root is colored black.
3) A red node can have only black children.
4) If a node of the BST does not have a left and/or right child then we add an external node.
5) External nodes are not colored.
6) The black depth of an external node is defined as the number of black ancestors it has.
7) In a RBT every external node has the same black depth.

Examples of RBTs :-

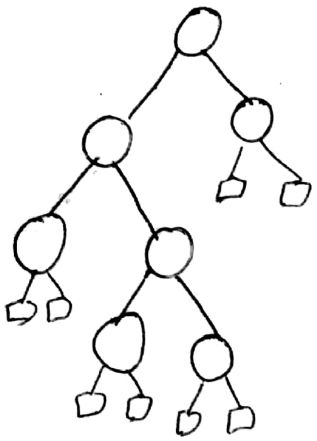


Black height of tree is 2

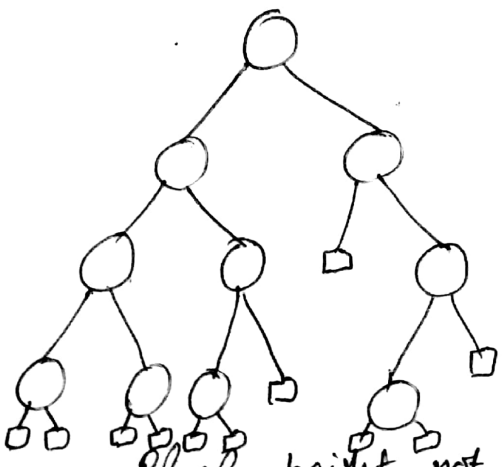


Black height of tree is 2.

Trees which are not red-black :-



Double red



Black height not uniform

Insertion - Possibilities :-

Things to remember :-

1) Z is always red during insertion.
2) After operation → New Z is the lowest node in violation.
3) Priority is given to parent than child during rotation for attachement.

Case 1 :-

Z is head - will always be black.

Rest of the cases depend upon uncle's mood

Case 2 :-

Z. uncle is red

→ Change colour of Z. Parent, Z. Uncle, Z. Grand Parent.

Case 3 :-

Z. Uncle is black and Z makes a triangle.

→ Rotate Z. Parent in opposite direction.

Case 4 :-

Z. uncle is black and Z makes a line.

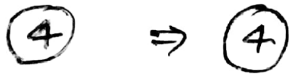
→ Rotate Z. GrandParent in opposite direction.

→ Exchange colours of Z. GrandParent and Z. Parent.

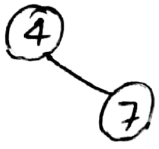
Example :-

4, 7, 12, 15, 3, 5, 14, 18, 16, 17

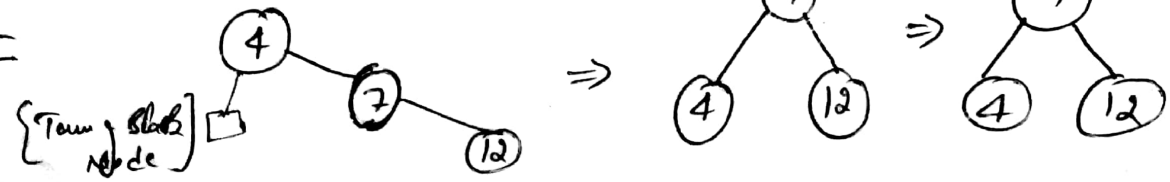
First 4 :-



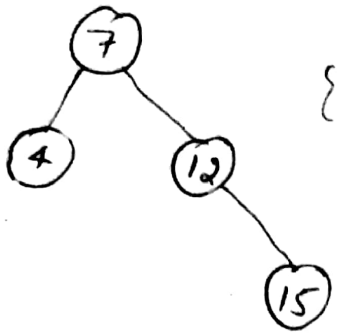
7 :-



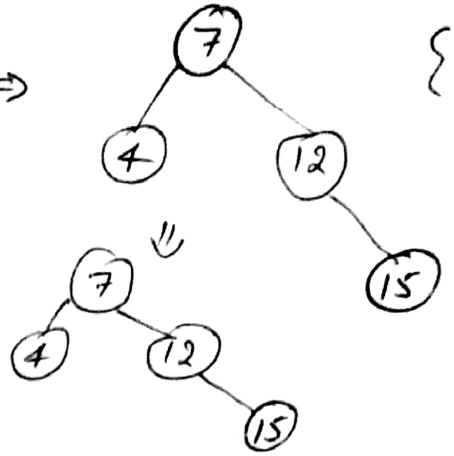
12 :-



15 :-



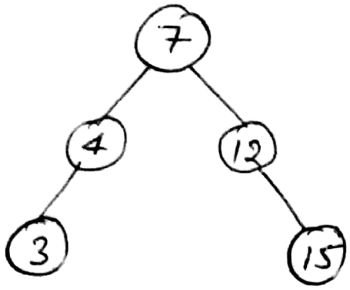
{ Case 2 } ⇒



⇓

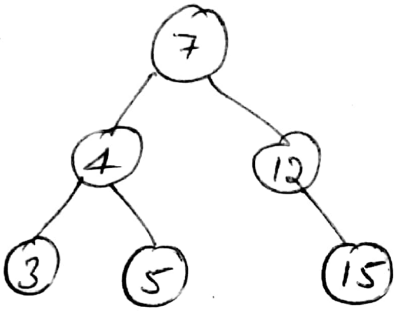
{ Case 1 must fulfill, So we say that Root is always black }

3 :-

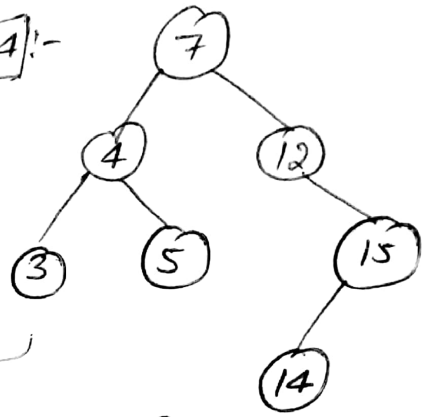


{ If there is no violation then do not see the properties }

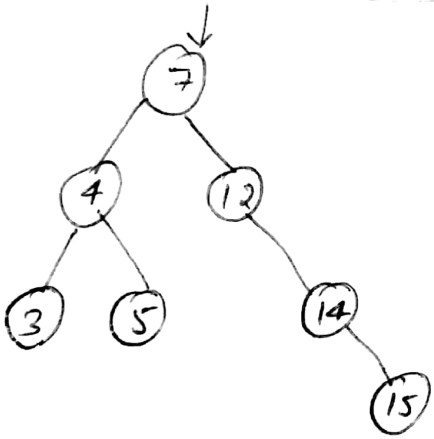
5 :-



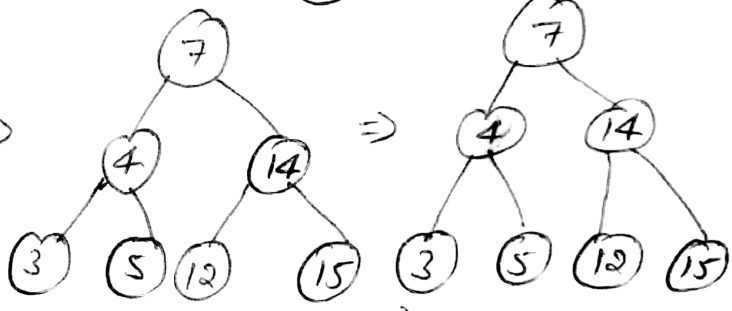
14 :-



{ Case 3 }

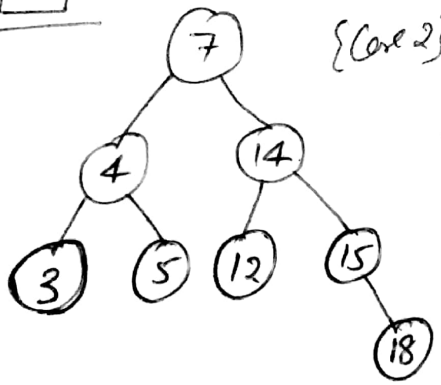


{ Case 4 } ⇒

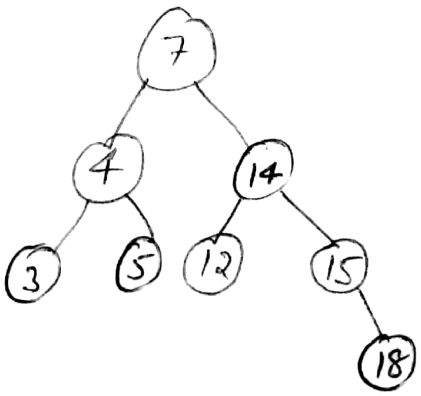


{ Rotate & Change color of 14 & 12 }

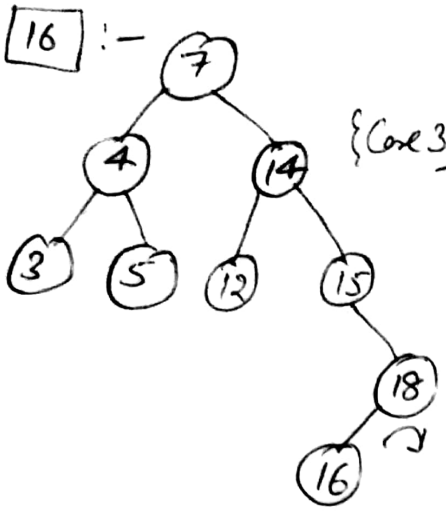
18 :-



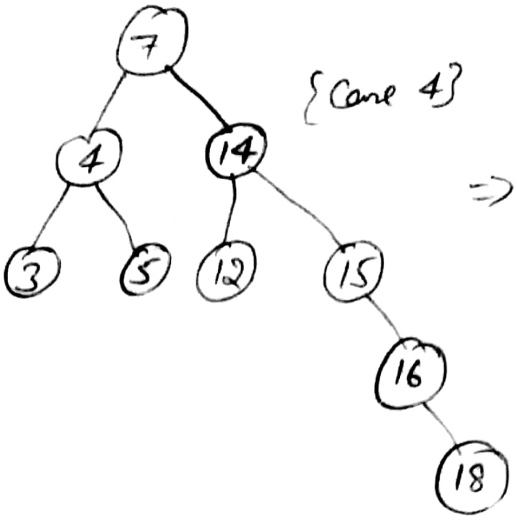
{ Case 2 } ⇒



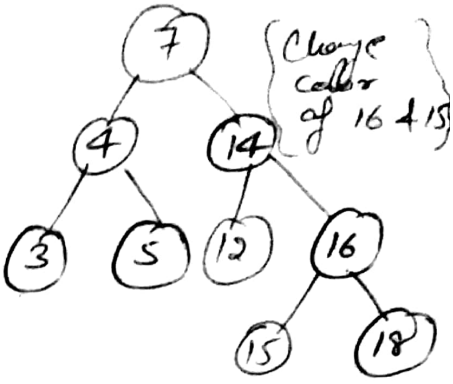
16 :-



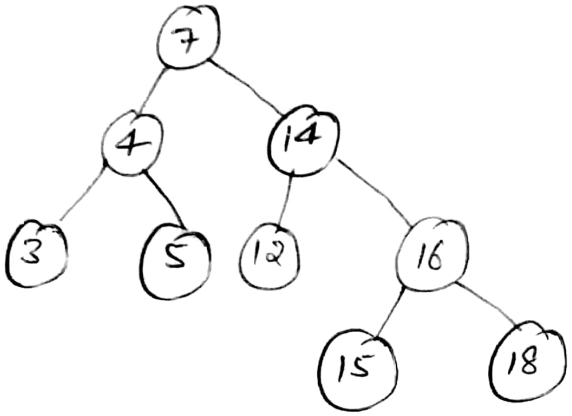
{Case 3} ⇒



{Case 4} ⇒

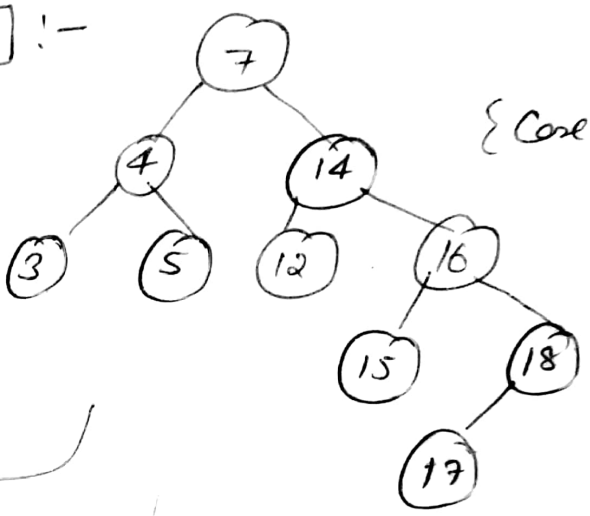


{Change color of 16 & 15}

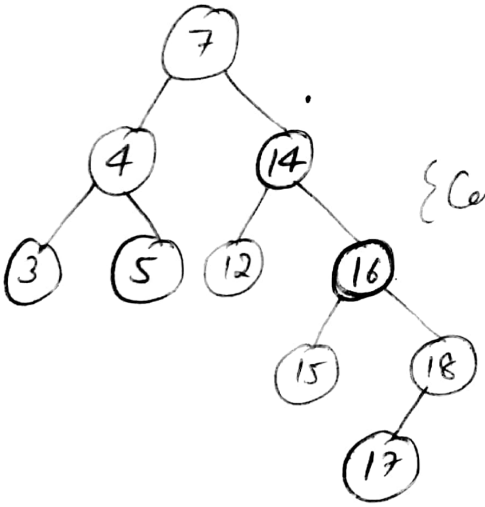


⇒

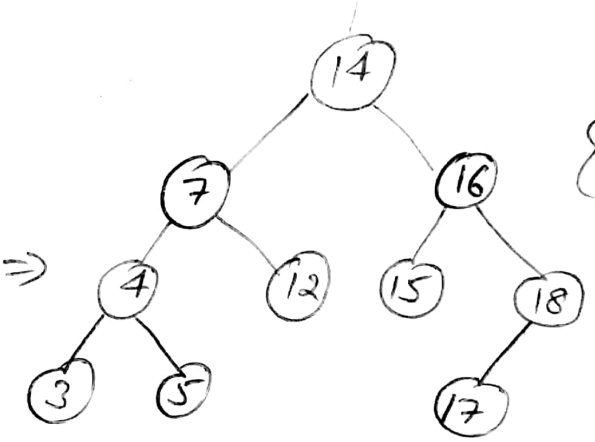
17 :-



{Case 2}



{Case 4} ⇒



{Priority will be given to Parent. Here 12 is child so give chance to 14}