[ALGORITHMS]

① P001

ALGO	PROGRAM
⊛ Written at Design time	⊛ Implementation time.
⊛ Domain Knowledge	⊛ Programmer
⊛ Any language	⊛ Programming language.
⊛ HW/OS independent	⊛ HW/OS dependent.

Analyze Algorithm :-

1. Time 2. Space 3. Network Consumption
4. Power Consumption 5. CPU Register [Device Drivers]

```
Algo swap(a, b) {            ⊛ Each step take SINGLE
                                 Time Unit
SPACE      tmp = a   → ①
  1 ←      a = b     → ①         f(n) = 3 = Constant
  1 ←      b = tmp   → ①                    value.
  1 ←  }                           ↓
                            Constant Denoted
S(n) = 3                          O(1)
 ↳ Again a constant
   space value.
```

Q. Find time complexity

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

No of times i RUN	No. of times j RUN	No. of times STMT RUN
0	0	0
1	0 ✓ 1 X	1
2	0 ✓ 1 ✓ 2 X	2
3	0 ✓ 1 ✓ 2 ✓ 3 X	3
⋮	⋮	⋮
n	0 1 2 ⋮ n	(n-1)

So total Number of times [stmt] runs will be $1+2+\cdots+(n-1) = \frac{(n-1)(n)}{2} = f(n)$

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

i	p
1	1
2	1+2 = 3
3	3(1+2)+3 = 6
4	6(1+2+3)+4
k	1+2+3+4+..k

Assume $p > n$

$p = k(k+1)/2 > n$ [STOP]

$\approx k^2 > n$

$\approx k > \sqrt{n}$

So order is $\sqrt{n}$

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

i	stmt RUNS
1	1 = 2^0
1x2	2
2x2	2^2
2^2x2	2^3
⋮	⋮
2^k	2^k

Assume $i >= n$

$2^k >= n$

which → $2^k = n$

$k = \log_2 n$ time complexity $O(\log_2 n)$

$n = 8$

i
1
2
4
8 (X) stop

3 times

$n = 10$

i
1
2
4
8
16 (X) stop

4

$\log_2 8 = 3$

$\log_2 10 = 3.2$ [assume]

when $n = 8$, loop runs = 3 times

when $n = 10$, loop runs = 4 times

hence log value in $O(\log_2 n)$ should be taken as CEIL. $O(\lceil \log_2 n \rceil)$

Q. p=0

```
for(i=1, i<n; i=i*2)
{   p++;
}   p - log(n)
```

```
for(j=1; j<p; j=j*2)
{   stmt;
}
```

stmt: → (log p)

stmt: is an order of log p & p is in order of log n → hence total order will be O(log log (n))

Q.

```
for(i=0; i<n; i++)  ⟶ n
{   for(j=1; j<n; j=j*2)   ⟶ total
    {   stmt;   ⟶ (log n)
    }
}
```

we got n times log n

that is O(n log n)

Q. P003

for(i=0; i<n; i++) ⟶ O(n)

for(i=0; i<n; i=i+2) ⟶ $\frac{n}{2}$ ⟶ O(n)

for(i=n; i>1; i--) ⟶ O(n)

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

for(i=1, i<n; i=i*2) ⟶ $O(\log_2 n)$

for(i=1; i<n; i=i*3) ⟶ $O(\log_3 n)$

for(i=n; i>1; i=i/2) ⟶ $O(\log_2 n)$

Analysis of if & While [Condition & Loop]

for(i=0; i<n; i++) → executes (n+1) times

{
  stmt → n times.
}

while(condition)
{
  stmt;
}

do
{
  stmt;
}
while(condition)

Q.

i=0
while(i<n) → executes (n+1) times
{
  stmt; → n times
  i++; → n times.
}

Q. a=1
while(a<b)
{
  stmt;
  a = a*2
}

↑ depends on "b" lets find out.

a
1
1x2=2
2x2=2^2
!
2^k

Terminate when

$a \geq b$

$2^k = b$

$k = \log_2 b$

Order is $O(\log n)$

Q0 $i=1, k=1$

P004

```
while(k<n)
{
    stmt;
    k=k+i
    i++;
}
```

i	k
1	1
2	1+1=2
3	2+2
4	2+2+3
5	2+2+3+4
!	!
m	2+2+3+4+---+m

Reduce ↓ Roughly

$1 \approx \frac{m(m+1)}{2}$

$k \geq n$

$\frac{m(m+1)}{2} \geq n$

$\approx m^2 \geq n$ → time complexity $O(\sqrt{n})$

↳ analogy for Statement

~~for(i=0;~~ ~~for(k=i, k<n, i++)~~

```
for(k=1, i=1; k<n; i++)
{
    stmt;
    k=k+i;
}
```

Q.

```
while( m != n )
{   if(m>n)
        m = m-n;
    else
        n = n-m;
}
```

m = 6	n = 3	
3	3	1 time
m = 5	n = 5	
0	0	1 time
m = 16	n = 2	
14	2	
12	2	
10	2	7 time
8	2	
6	2	
4	2	
2	2	stop

min O(1)
max O(n)

Q.

```
ALGO Test(n)
{   if(n<5)
    {   print("%d", n)  → 1 (time) [Best case]
    }
    else
    {   for(i=0; i<n; i++)
        {   printf("%d", n)  [n time] → Worst case.
        }
    }
```

[Best O(1)
Worst O(n)]

Types of Time Complexity

$O(1)$ → constant → $f(n)=2$, $f(n)=500 \rightarrow O(1)$

$O(\log n)$ → Logrithimic

$O(n)$ → Linear $f(n)=2n+3$, $f(n)=500+n+7$ → $O(n)$

$O(n^2)$ → Quadratic

$O(n^3)$ → Cubic

$O(2^n)$ → Exponential

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

$\ldots < O(n^2), O(n^3) \ldots O(2^n) \ldots$

Increasing Order of Time Compleaity.

Asymptotic Notation

O - big-oh [Upper Bound]	Ω - big-omega [Lower Bound]	Θ - theta [Average Bound]

Best, Worst, Average Case Analysis

Explained by taking [Linear Search] as example.

A

8	6	12	5	9	7	4	3	16	18
0	1	2	3	4	5	6	7	8	9

Best case searching for key present at index = 0 [first index]

Best case will be order $O(1)$

Worst Case : Maximum time, key at last index.

Worst case :- time is [n] order of $O(n)$.

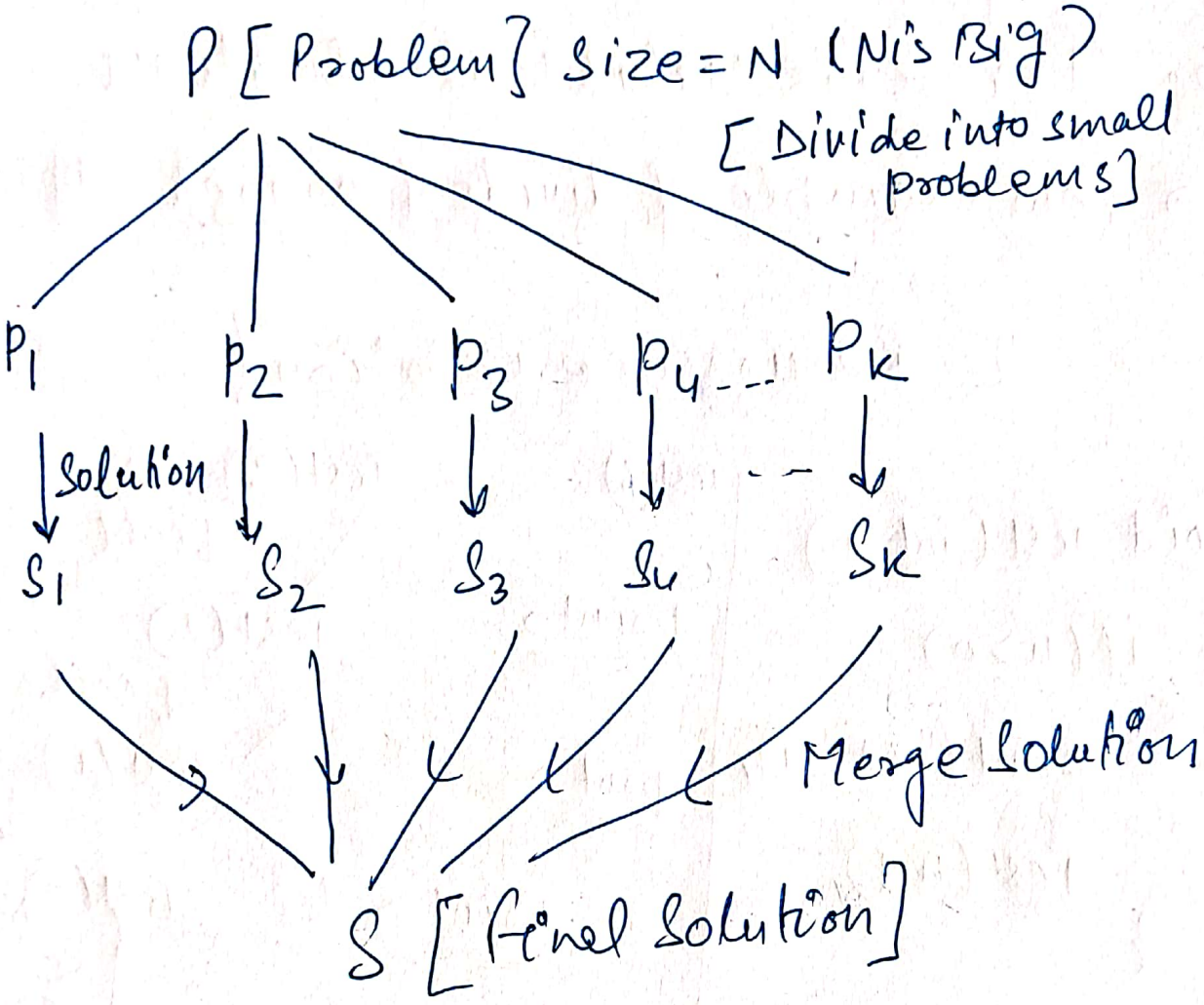
Average Case :- $\frac{\text{All possible Case time}}{\text{No. of cases}}$

(*) This analysis not possible at every situations so most of the time we do worst case analysis.

$$\text{Avg time} = \frac{1+2+\cdots n}{n} = \frac{n(n+1)}{2*n}$$

$$= \left(\frac{n+1}{2}\right) \text{ order is } \rightarrow O(n)$$

Divide & Conquer



(*) While dividing problem it has to be same

Ex: P → Sort an big array.

$P_1, P_2, \ldots P_k$ → All should be the sorting problem.

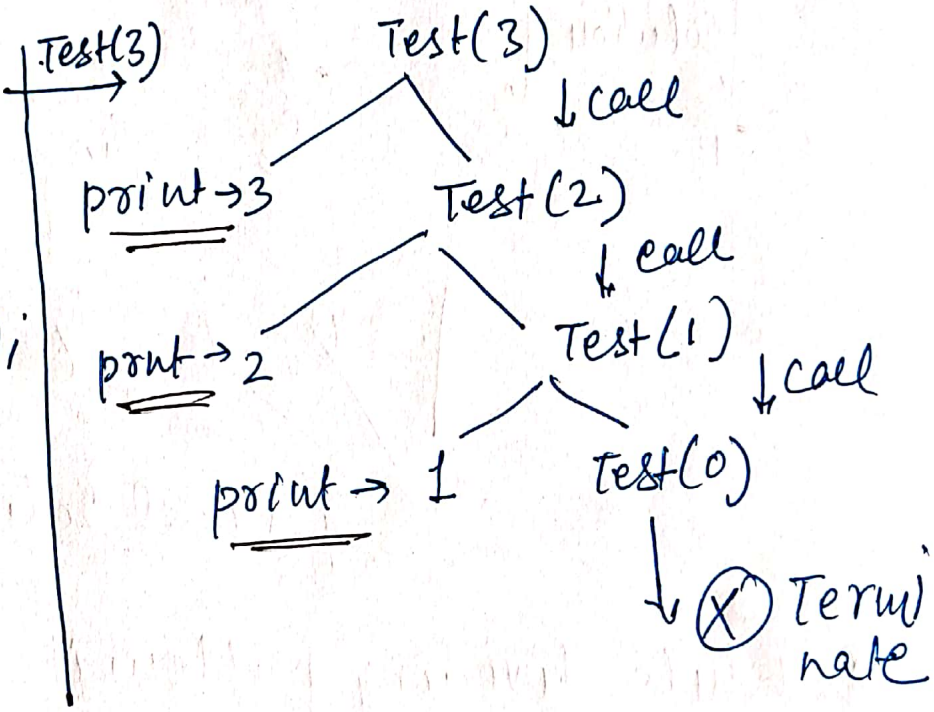
(*) We should have method for combining all "SUB-SOLUTION"

1. Binary Search 2. Find Max & Min
3. Merge Sort 4. Quick Sort
5. Strassen's Matrix Multiplication

↳ few example of divide & conquer approach.

[Recurrence Relation]

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



(A) Above function printing value 3 times for Test(3) & calling itself 4 times.

To do for pass "n" of Test(n) → n times print.

Prepare Recurrence Relation

```
void Test(n)
{ if(n>0)
  { printf("%d",n)
    Test(n-1)
  }
}
```

→ Total amount of time let say = T(n)

printf → 1

Test(n-1) → T(n-1)

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

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

Solve :-

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

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

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

Substitute :-

$$T(n) = [T(n-1)+1] + 1$$
$$= T(n-2) + 2$$
$$\vdots \quad k \text{ times}$$
$$T(n-k) = T(n-k) + k$$

reach till $n-k=0$

$n=k$

$$T(n) = T(n-n) + n$$
$$T(n) = T(0) + n = 1 + n$$

```
void test(int n)
{ if(n>0)
  { for(i=0;i<n;i++){
      print(n)
    }
    test(n-1)
  }
}
```

say → T(n)

if → 1

for → n times

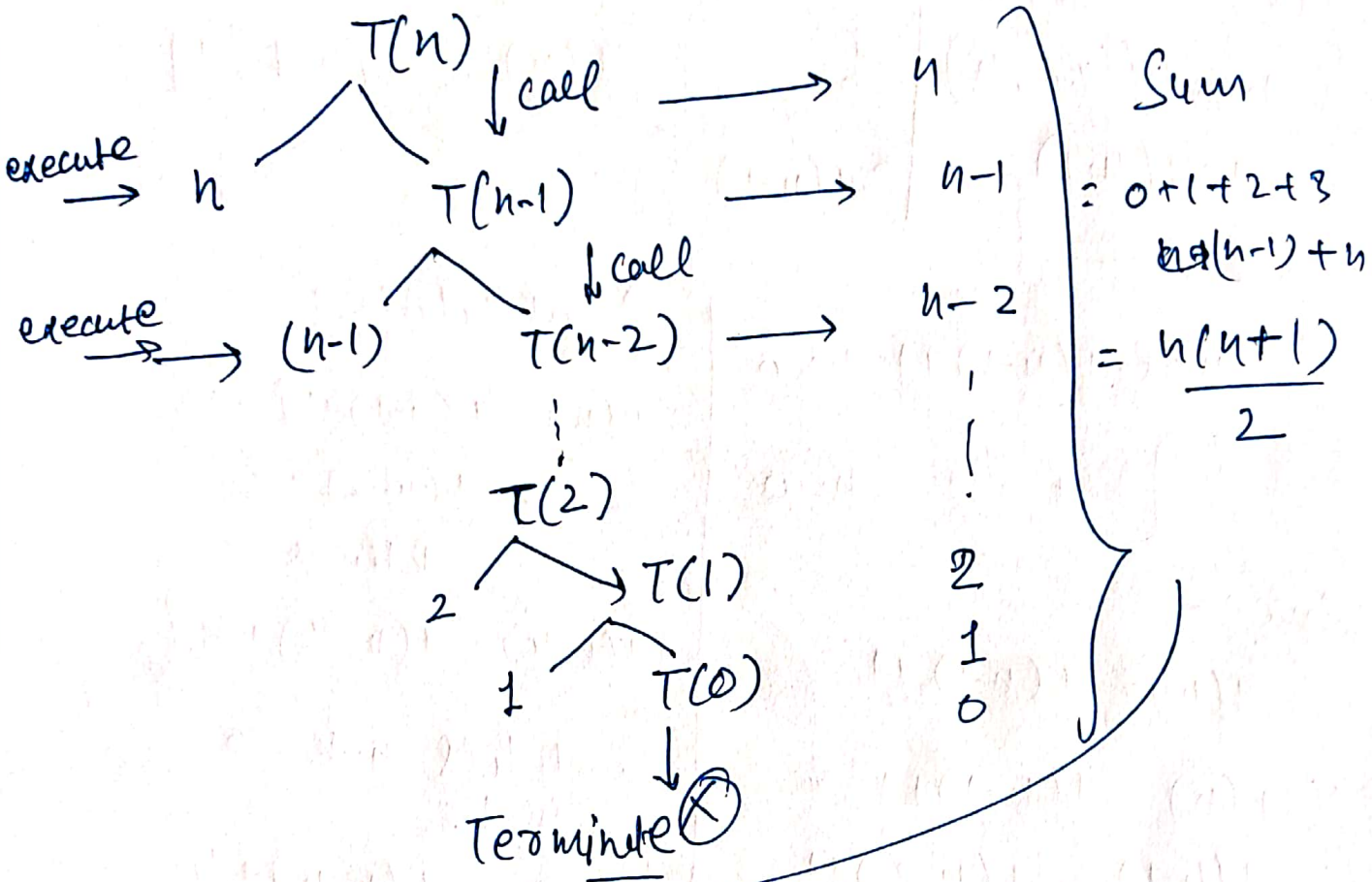
test(n-1) → T(n-1)

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

↓ linear

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

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



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

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

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

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

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

! k times

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

assume that we reach $n-k = 0$

$\boxed{n = k}$ substitute

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

$T(n) = T(0) + 1 + 2 + 3 + 4 + \cdots + n$

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

P008

```
void test(int n){
  if (n>0){
    for(i=0, i<n; i=i*2){
      print(i);
    }
    Test(n-1);
  }
}
```

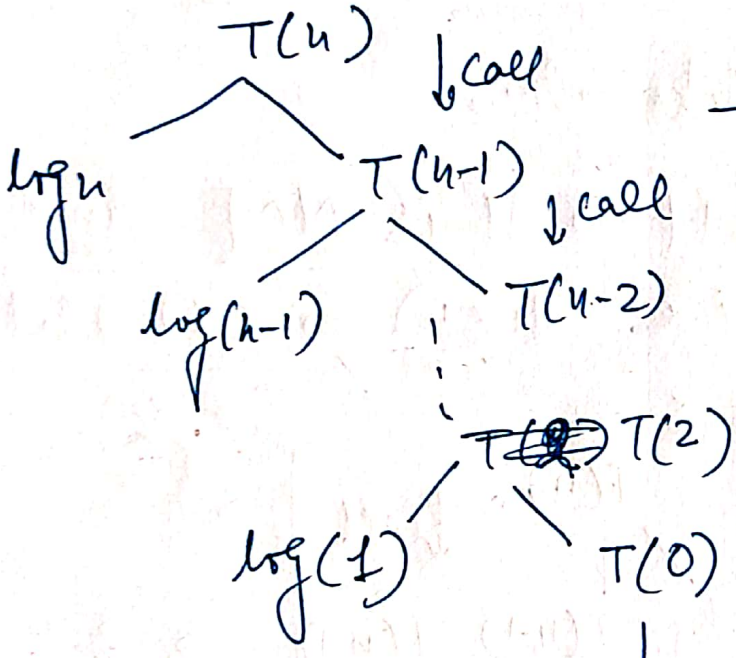
$T(n)$

$\log n$

$T(n-1)$

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

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



Total Time =

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

$\log(n \times (n-1) \times (n-2) \dots \times 2 \times 1)$

$= \log(n) = \log(n!)$

$\hookrightarrow O(n \log n)$

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

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

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

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

when $n-k=0$

$n=k$

$T(n) = T(0) + \log n!$

$O(n \log n)$

can be imagine as below

$T(n) = T(n-1)+1 \rightarrow O(n) \rightarrow n \times 1 = $ Solinear

$T(n) = T(n-1)+n \rightarrow O(n^2) \rightarrow n \times n = So[n^2]$

$T(n) = T(n-1)+\log n \rightarrow O(n\log n) \rightarrow n \times \log n = n\log n$

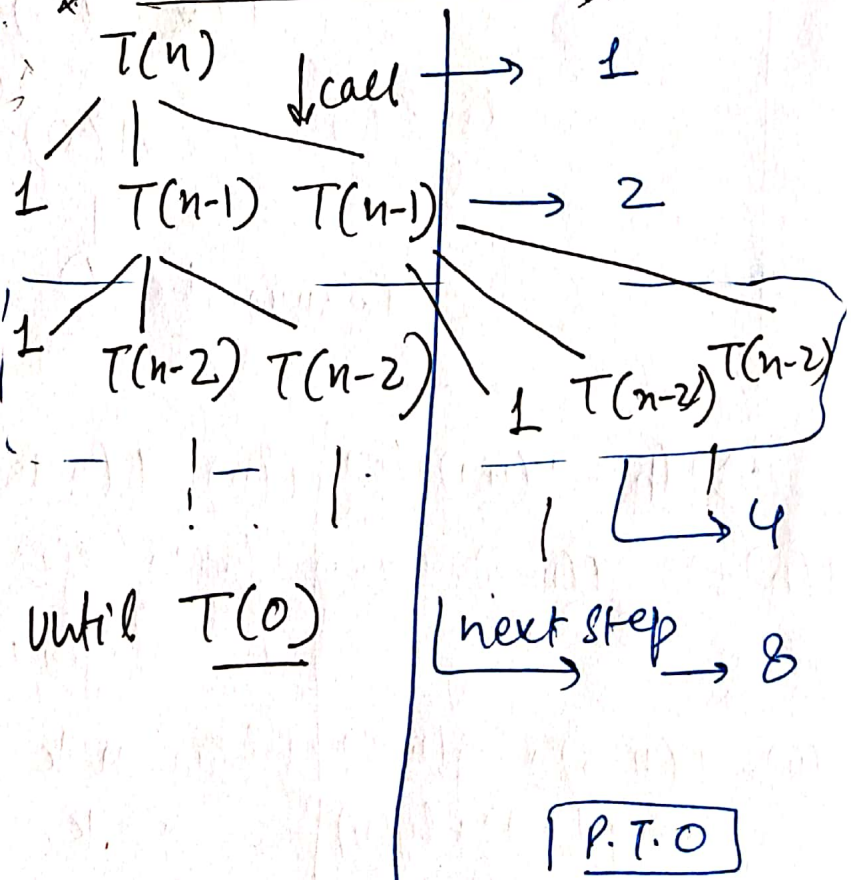
$T(n) = T(n-1)+n^2 \rightarrow O(n^3) \rightarrow n \times n^2 \rightarrow O(n^3)$

but if we have like below

[$T(n) = 2T(n-1)+1 \rightarrow$ It is NOT same!! as above i.e $(n \times 1) \rightarrow O(n)$ let see.

```
Algo Test(int n) {→T(n)
  if(n>0) {
    print(n); — 1
    Test(n-1) ;→T(n-1)
    Test(n-1);→T(n-1)
  }
}
```

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



P.T.O

$1, 2^1, 2^2, 2^3 \dots 2^k$

Total time $= 1 + 2^1 + 2^2 + 2^3 \dots 2^k$

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

Assume $k-(k) = 0 \quad n = k$ order of $O(2^n)$

$T(n) = 2T(n) + 1$ belongs to $O(2^n)$

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

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

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

$T(n) = 2^k T(n-k) + 2^{k-1} + 2^{k-2} + 2^2 + 2^0 \dots$

→ When $n-k=0 \rightarrow n=k$

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

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

$\rightarrow O(2^n)$

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

$\hookrightarrow O(2^n)$

$T(n) = 2T(n-1) + n \rightarrow O(n \times 2^n)$ [Master Theorem Decreasing Function]

$T(n) = a * T(n-b) + f(n)$

assume that $a > 0, b > 0$, & $f(n) = O(n^k) \quad k \geq 0$

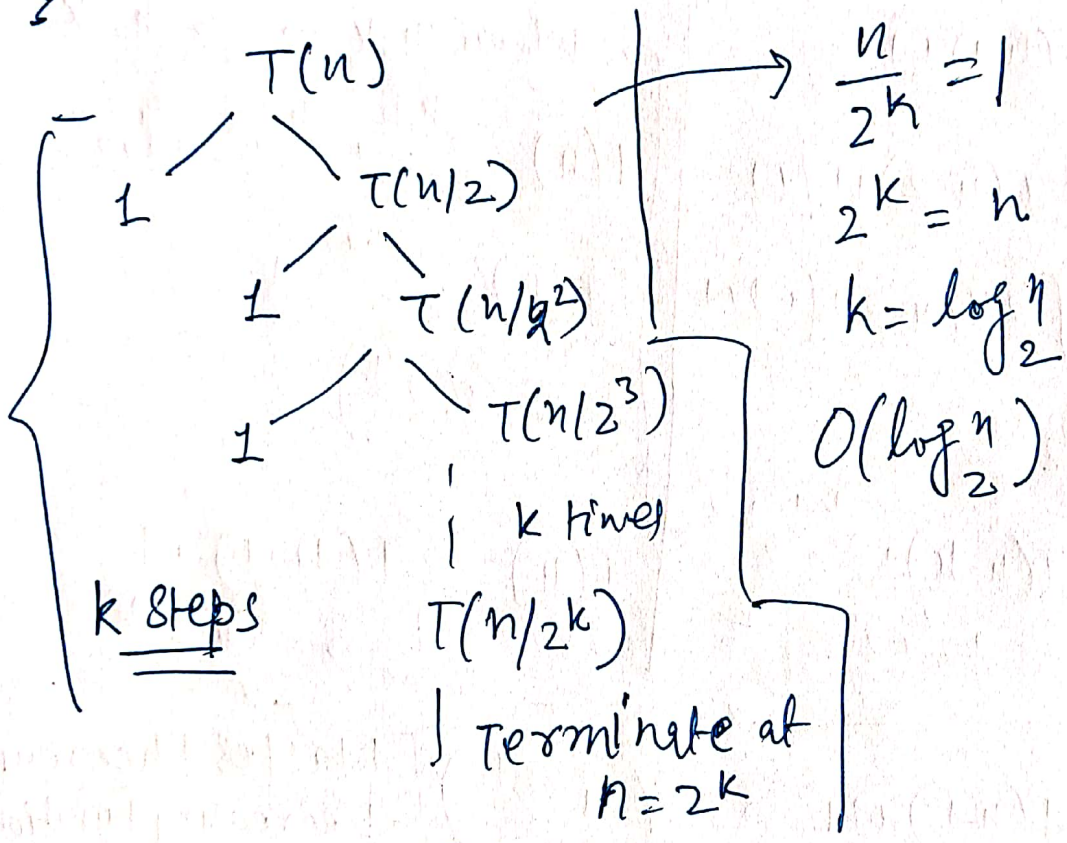
if $a = 1$	if $a > 1$	if $a < 1$
$O(n^{k+1})$ or $O(n * f(n))$	$O(n^k a^{n/b})$	$O(n^k)$ or $O(f(n))$

Dividing function

```
Algo Test(int n){          → T(n)
    if(n>1) {
    print(n);              → 1
    Test(n/2);             → T(n/2)
}}
```

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

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



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

$2^k = n$

$k = \log_2 n$

$O(\log_2 n)$

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

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

$\vdots$

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

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

$n = 2^k \rightarrow k = \log n$

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

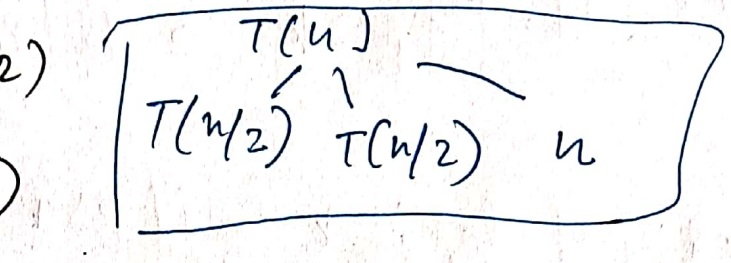
$= 1 + \log(n)$

$O(\log n)$

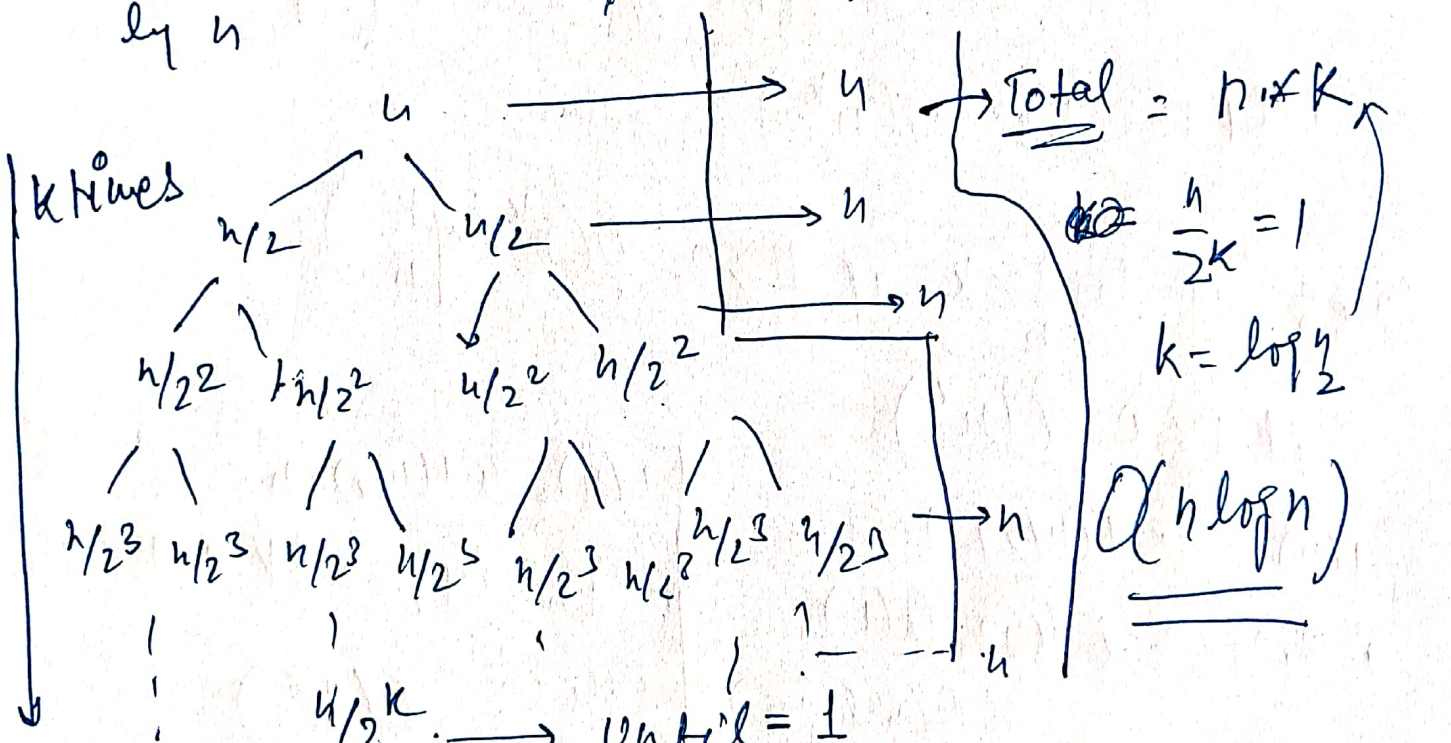
P010

```
void Test(int n) {          → T(n)
  if(n>1){
  for(i=0; i<n; i++){
  stmt;
  }
  Test(n/2);    → T(n/2)
  Test(n/2);    → T(n/2)
}
```

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



avoiding calling T(n) lets write only n



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

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

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

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

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

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

↳ until $\Rightarrow \frac{n}{2^k} = 1$

$k = \log_2 n$

$2^k * 1 + n\log n$

$n + n\log n$

$O(n\log n)$

Masters Theorem for Dividing function :-

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

$f(n) = \Theta(n^k \log^p n)$

case 1: if $\log_b^a > k \rightarrow \Theta(n^{\log_b^a})$

Case 2: if $\log_b^a = k \rightarrow$ if $p > -1$ $\Theta(n^k \log^{p+1} n)$

if $(p = -1)$ $\Theta(n^k \log\log n)$

if $(p < -1)$ $\Theta(n^k)$

Case 3: if $\log_b^a < k$ if $p \geq 0$ $\Theta(n^k \log^p n)$

if $p < 0$ $\Theta(n^k)$

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

$a = 2, b = 2 ; f(n) = \Theta(1)$

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

↳ $k = 0, p = 0$

$\log_b^a = \log_2^2 = 1$ → Case 1

$\Theta(n^1)$

$T(n) = 4\,T(n/2) + n$

$\log_b^a = \log_2^4 = 2 \mid k = 1, p = 0$

case 1: $\Theta(n^2)$

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

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

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

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

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

$\vdots$

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

k times

Assume $n = 2^m$

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

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

assume $T\left(2^{\frac{m}{2^k}}\right) = T(2)$

$\frac{m}{2^k} = 1 \rightarrow k = \log_2 m$

Since $n = 2^m \rightarrow m = \log_2 n \rightarrow k = \log\log_2 n \rightarrow O(\log\log_2 n)$

P.T.O

Binary Search Iterative :-

A =

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
3	6	8	12	14	17	25	29	31	36	42	47	53	55	62

↑ low = l

key = 42

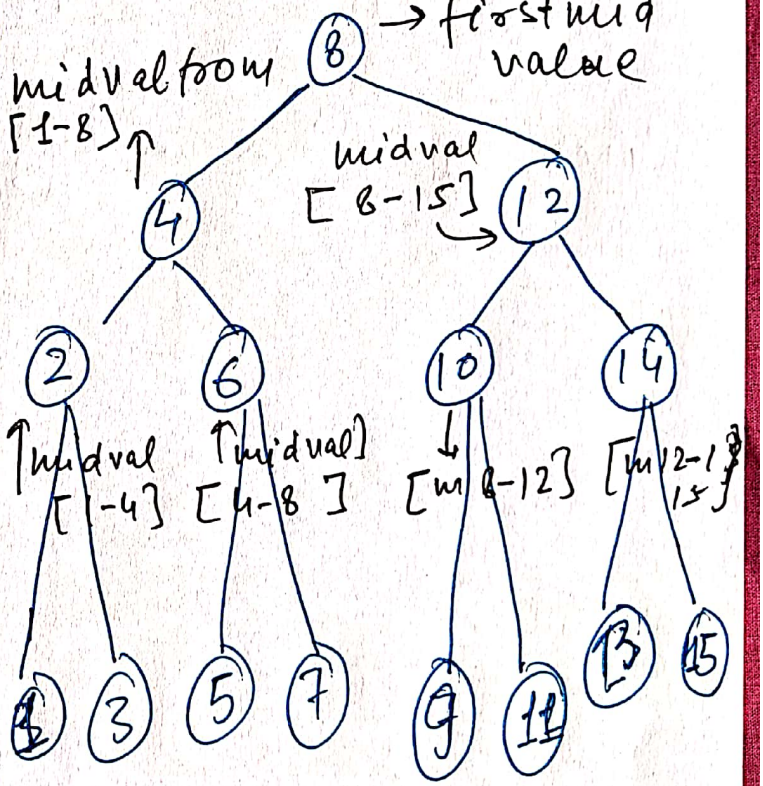
↑ high = h

$mid = \lfloor \frac{l+h}{2} \rfloor$ → floor value

$= \frac{1+15}{2} = 8$

→ Visualizing With Tree mid values

```
int BinarySearch(A, n, key){
  l = 1, h = n
  mid = [(l+h)/2]
  while(l <= h) {
  mid = (l+h)/2
  if(key == A[mid])
    return mid;
  if(key < A[mid])
    h = mid-1
  else
    l = mid+1
  }
  return -1
}
```



Above is a Tree how program jump on my Array [A] & tree form visualization.

(K) Binary Search Recursive :-

Binary Search is Divide & Conquer type. Divide & Conquer we say if problem is large divide into smaller problem.

So we should define [WHAT IS SMALL PROBLEM?] Then only we can say other are large. Here we are saying size Arr=15 is large. So define SMALL !

So I say if Arr=1 = Single element then it is SMALL

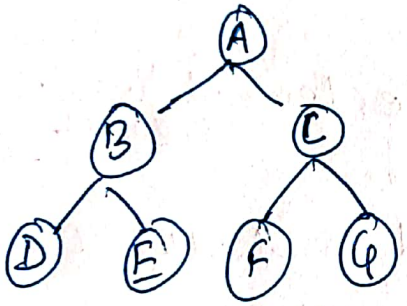
```
Algo RecBinarySearch(l, h, key){
if(l==h){// In this case Arr=1 single element hence solve it //
   if(A[l]==key)
      return l;        } key element found for SMALL PROBLEM defined above & found.
   else
      return -1;       } key element not found
}⇒ SMALL PROBLE HANDLED
else{ ⇒ Problem is large
mid = (l+h)/2;
if(A[mid]==key)
      return mid;
if( key < A[mid]) → Break problem RecBinary Search on left side.
   return RecBinaseSearch(l, mid-1, key);
else → Rec Binary Search on Right side
   return ReBinasSear(mid+1, h, key);
}}
```

Heap [Heap]

Topics Covered :-

1. Array Representation of Binary Tree
2. Complete Binary Tree 3. Heap 4. Heap Sort
5. Heapify 6. Priority Queue.

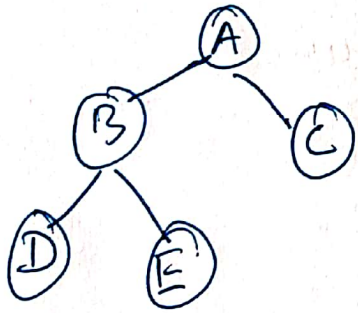
1. Array Representation of Binary Tree :-



→ T

A	B	C	D	E	F	G
1	2	3	4	5	6	7

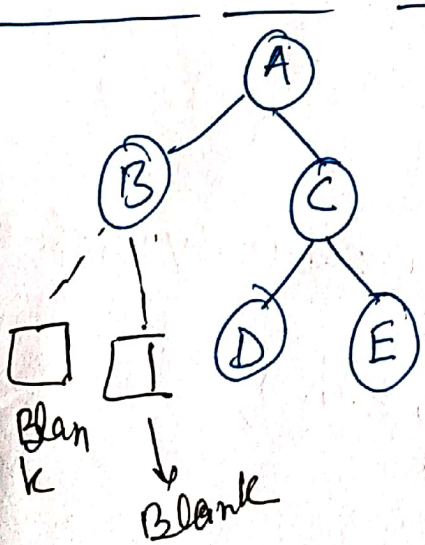
Ⓐ Node at index i
→ left child at 2*i
→ right child at 2*i+1
→ parent at $\lfloor i/2 \rfloor$



→ T

A	B	C	D	E

Same as above



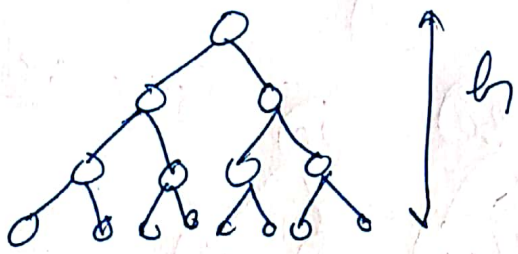
T →

A	B	C	-	-	D	E

Ⓐ if some child is missing put a blank these then only above formula will be applicable.

④ Complete Binary Tree

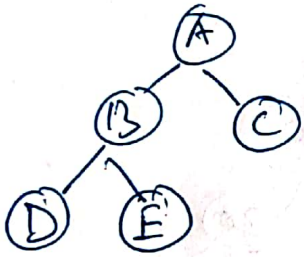
→ FULL Binary Tree :-



→ No space for any new node i.e it will increase the size of Binary Tree if appended.

max element with height(h) = $2^{h+1} - 1$

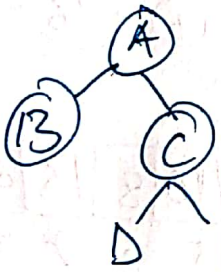
→ Complete Binary Tree :-



Represent in Arr →

A	B	C	D	E	

[NO GAP in between]



Represent in Arr →

A	B	C	_	_	D	E

GAP

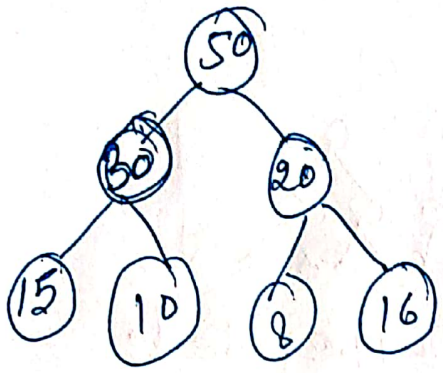
↳ Not Complete Binary Tree

④ Complete Binary of height (h) is FULL Binary Tree till (h-1) & last level elements are filled left to right.

⊛ Heap

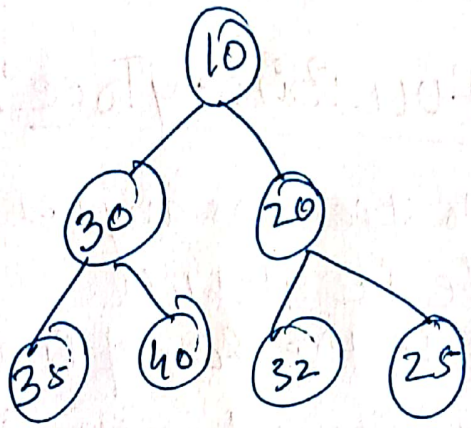
Heap is a Complete Binary Tree.

MaxHeap



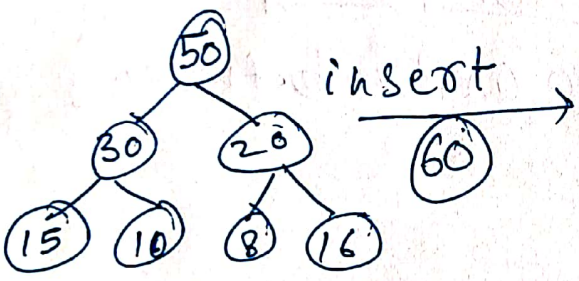
* Every node value ≥ all it's decendent

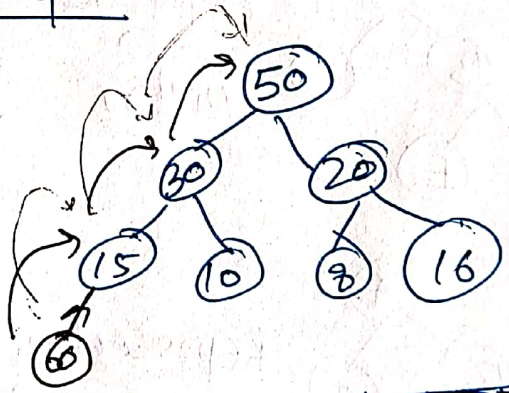
MinHeap



⊛ Every node ≤ all it's decendent.

⊛ Inser operation in MaxHeap :-





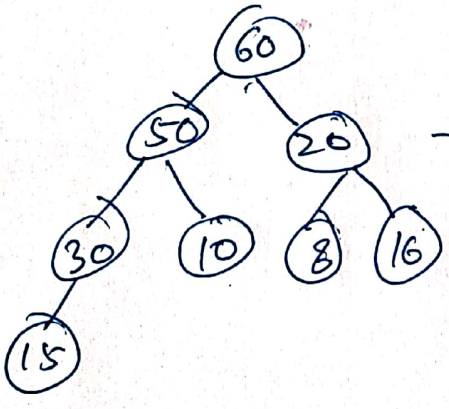
H

50	30	20	15	10	8	16
1	2	3	4	5	6	7

H→

50	30	20	15	10	8	16	60
1	2	3	4	5	6	7	8

H→ (60) compared to it's parent swapped until reached correct place.



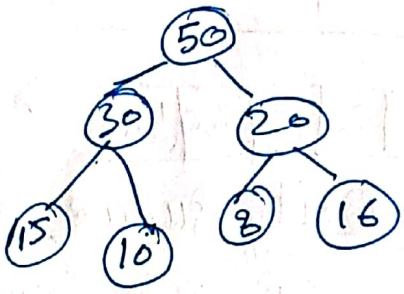
H→

60	50	30	20	15	10	8	16

Time complexity = O(log n)

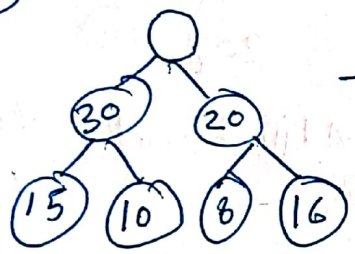
✱ Delete Operation Heap :-

H → | 50 | 30 | 20 | 15 | 10 | 8 | 16 |
1 2 3 4 5 6 7



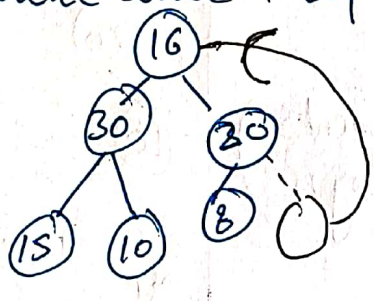
✱ Deletion can be perform only on ROOT element.

1.



H | | 30 | 20 | 15 | 10 | 8 | 16 |

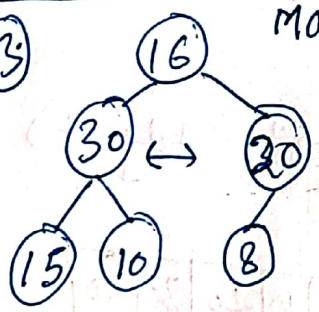
2. last elm in tree will come at 50 place



| 16 | 30 | 20 | 15 | ~~30~~ | 8 |

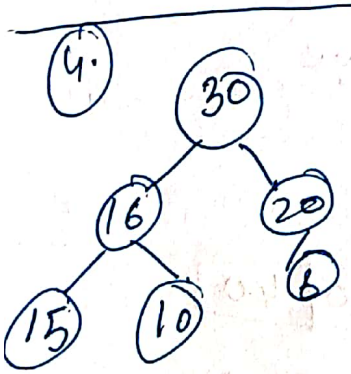
Maintain complete binary tree is important done now adjust element.

3.



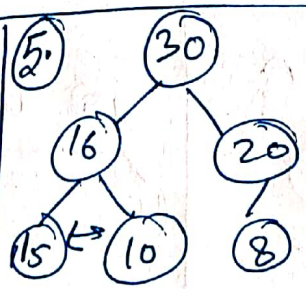
compare child (30, 20) greater will take it's place

4.



H → | 30 | 16 | 20 | 15 | 10 | 8 |

5.



(15, 10) compare
15 < 16 hence max heap property satisfy.

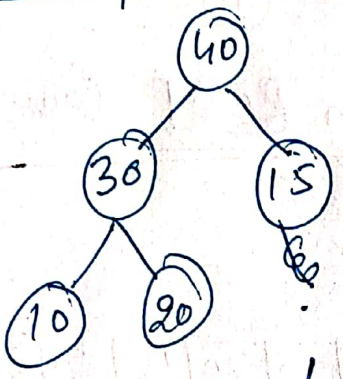
Deletion Time complexity O(log n)

Heap Sort :- **Step 1:** Create Heap from set of elements.

Step 2: Delete elements one by one.

10	20	15	30	40

create a priority queue max heap

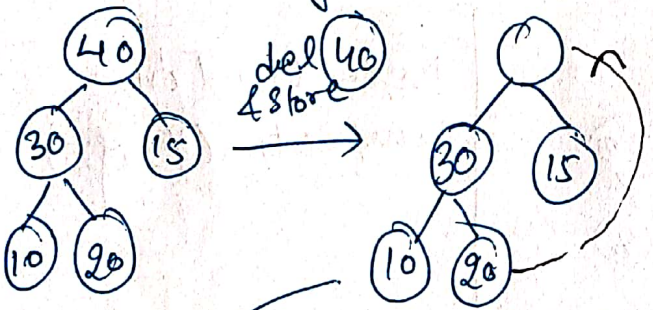


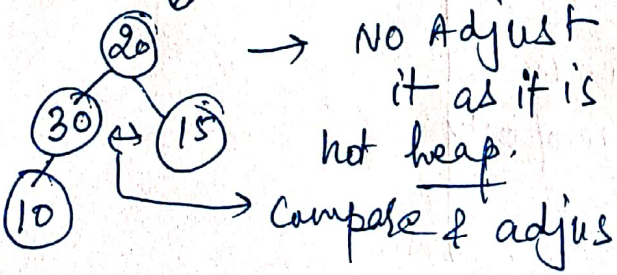
Total time taken = (n elements × log n)

= O(n log n)

H → | 40 | 30 | 15 | 10 | 20 |

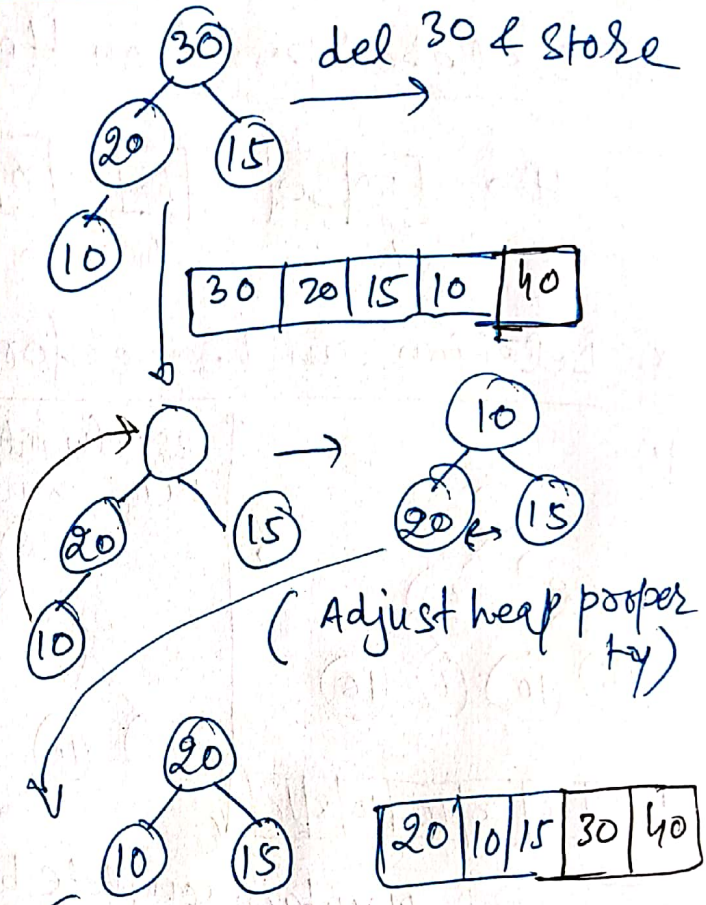
Deleting elements

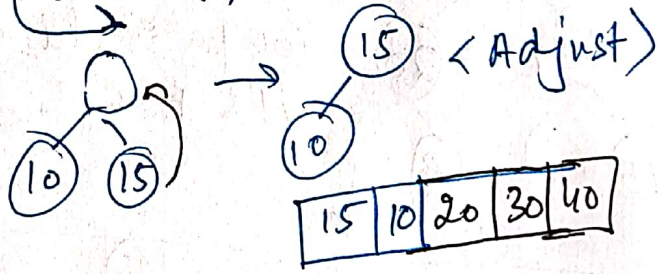


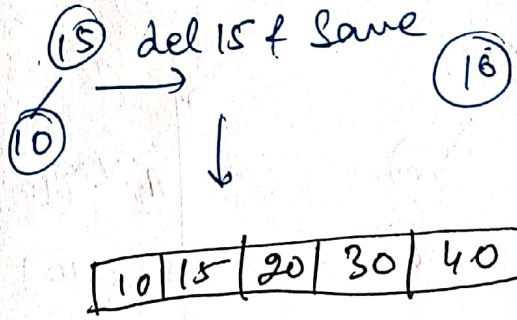


→ No Adjust it as it is not heap.

→ Compare & adjust





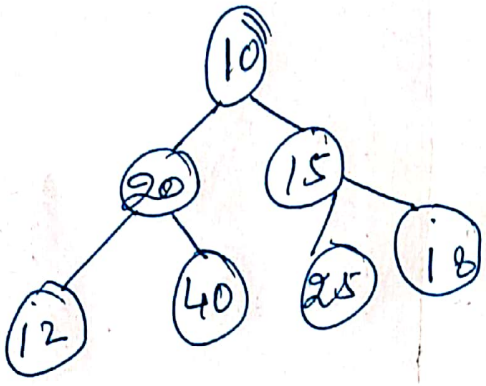


create heap → n log n

delete heap → n log n

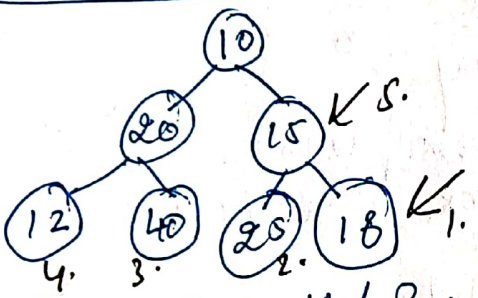
total time → O(n log n)

Heapify :-



H →

10	20	15	12	40	25	18



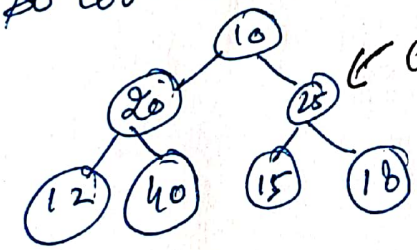
* Right to Left (R→L)

First check element (18) look down. No children

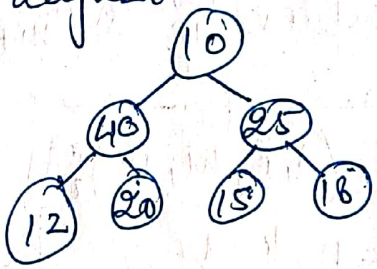
[DON'T LOOK ALL OTHER NODE (18) & it's decendent]

* 18, 25, 40, 12

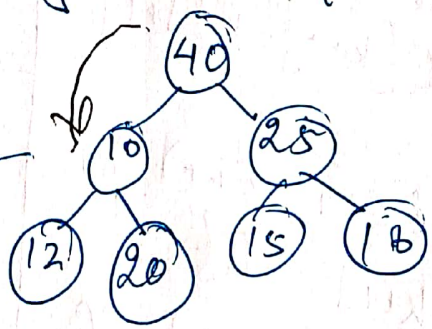
* Now (15), (15) has children compase, it's not a heap so correct it



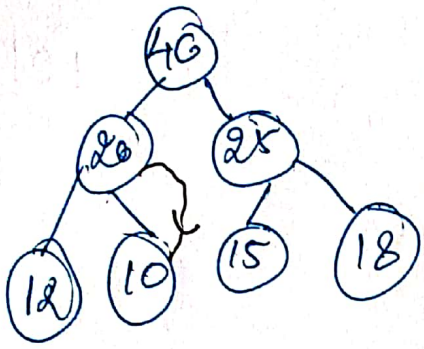
* element (20) look down -ward is it a ~~max Heap~~ max heap. NO so compare & adjust.



* element (10) look adjacent down ward & check is it a max-heap. NO. So adjust & compare.

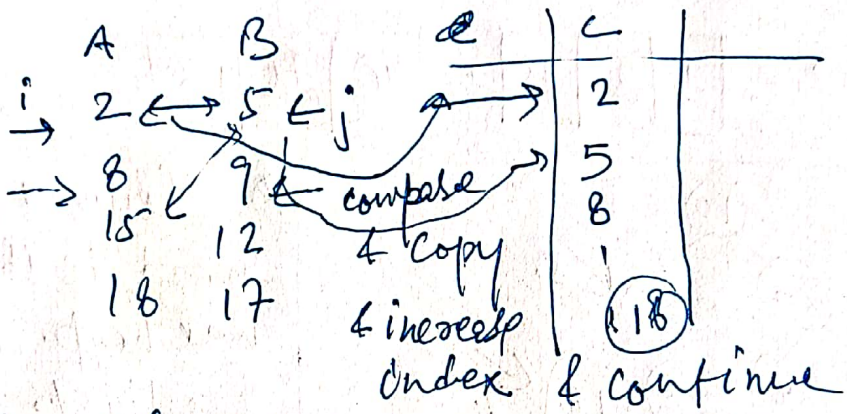


→ again compare with adjacent & correct it



Time complexity $O(n)$

Merge Sort :-



(*) Two sorted lists.

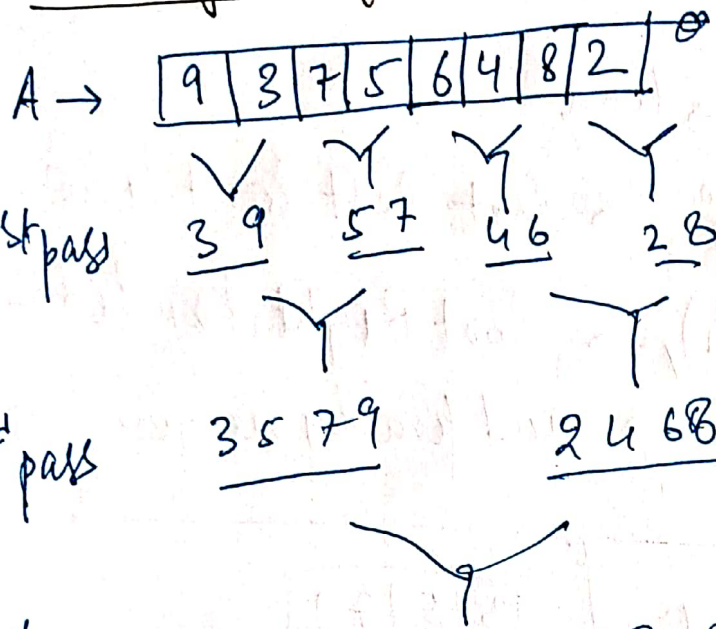
(*) Time complexity $O(m+n)$ Merging.

```
Algo_merge(A, B, m, n){
    i=0; j=0, k=0
while(i<m && j<n){
    if(A[i] < B[j]
        C[k++] = A[i++]
    else
        C[k++] = B[j++]
}
for(i i<m ; i++)
    c[k++] = A[i]
for(i j<n j++)
    c[k++] = B[j]
```

comparing & indexing or increasing of the array.

→ Copy remaining elements

ⓐ 2-Way Merge Sort :-



how many time $8 = 2^3$ = 3 pass $\lceil \log n \rceil$ times

so time complexity $O(n \log n)$.

ⓐ Merge sort Recursive :- Merge sort is divide & conquer. So when we have single elements then we say it is a small problem. ~~else~~ else it is a big problem.

Algo mergeSort(l, h)

if(l<h) {

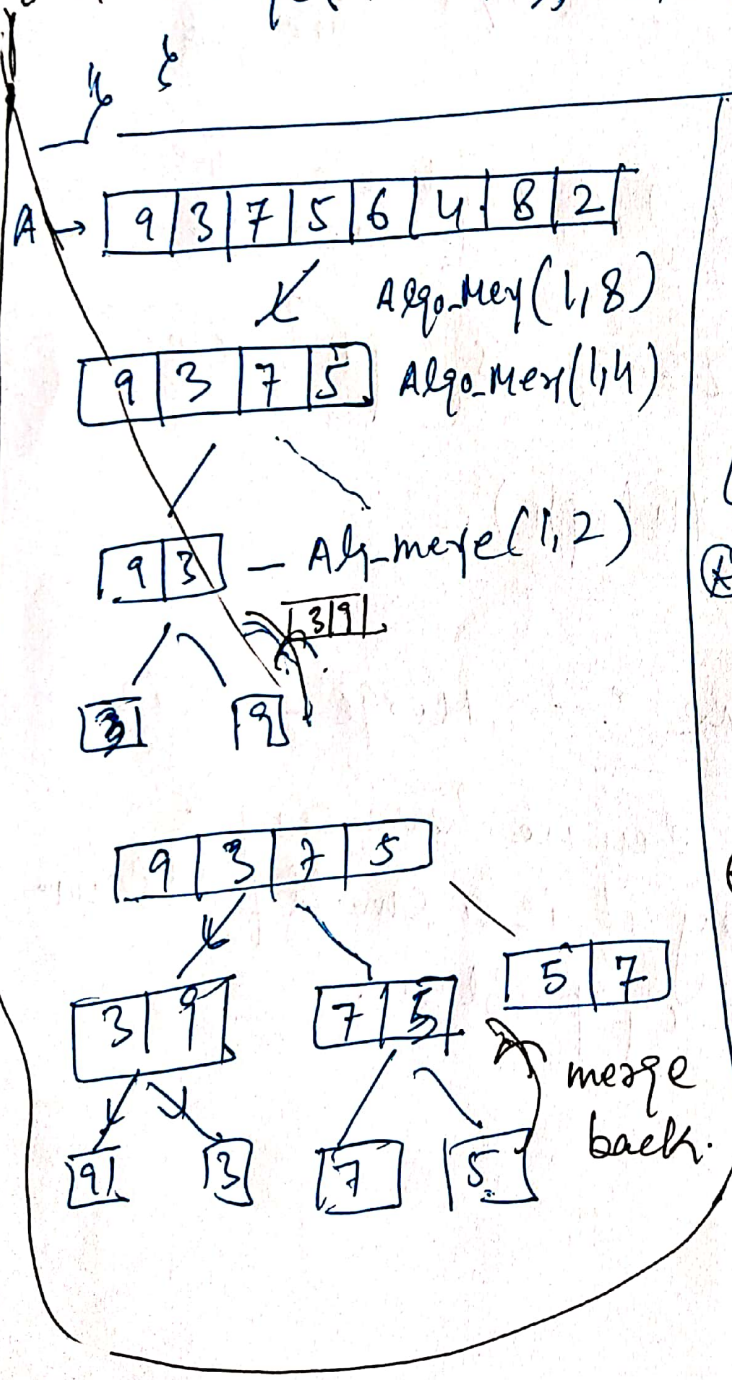
1 — mid = l+h/2 → find

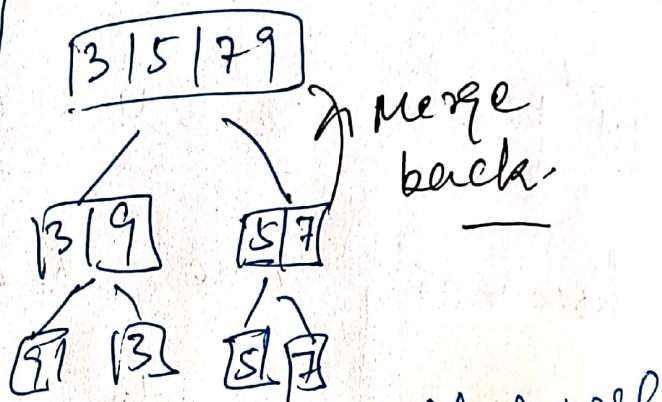
T(n/2) mergeSort(l, mid); → Sort left hand side

T(n/2) mergeSort(mid+1, h); → Sort Right hand side

n merge(l, mid, h); → and finally merge.

}





(*) Now it goes other part

6 4 8 2

Same Algo goes on.

(*) Recurrence Relation

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

$T(n) = 1 \quad n = 0$

⊛ Pros & Cons of Merge Sort

Pros: 1. Large Size List
2. Suited for Link List [because merging can be done by simple pointing correct node]
3. Suppor External Sorting :- (4) Stable Sort

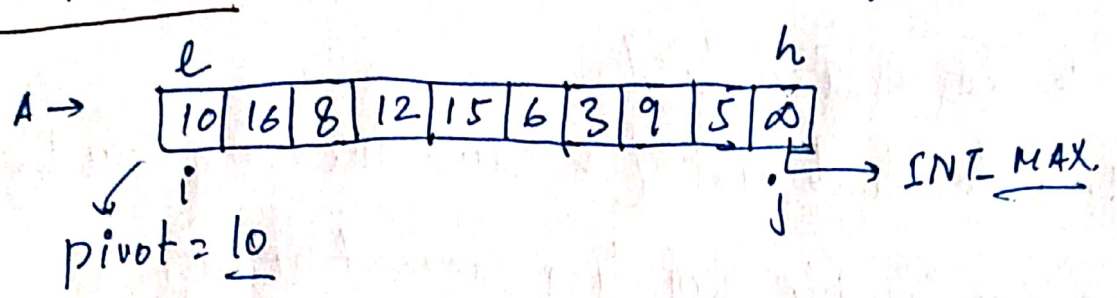
Con: Extra space (Not in place sort) [only for Arr] [not for Linklist]

⊛ Small size List not suitable. as compare to say (insertion Sort): It is obseved if ($n \leq 15$) then merge sort is not suitable.

⊛ Why insertion → [Also stable sort], Bubble Sort is also stable sort with $O(n^2)$ but insertion sort is suitable for Link List as Merge Sort.

⊛ Recursive: It uses system memory ⊛ (Stack memory) of $O(\log n)$

Quick Sort :- It is a divide & conquer method.



* Find the position of pivot element such [ALL element > pivot on RIGHT side, ALL element < pivot on LEFT side]

* Increment i, & decrement j, such that A[i] > pivot & A[j] < pivot & SWAP, by SWAP we are saying A[i] > 10 < A[j] so correct should b smaller should be on left of pivot & greater should be right side hence SWAP [It's like correcting position].

A → A[0]

10	16	8	12	15	6	3	9	5	∞

pivot, i, j

① increment i, until A[i] > pivot

10	16	8	12	15	6	3	9	5	∞

pivot, i

A[i] > pivot

② decrement j until A[j] < pivot

10	16	8	12	15	6	3	9	5	∞

A[j] < pivot

stop & SWAP.

10	5	8	12	15	6	3	9	16	∞

i

now
↳ increment i such that A[i] > pivot, then decrement

↳ SWAP A[j] & pivot.

6	5	8	9	3	10	15	12	16	∞

partition, pivot, partition

pivot (is at correct place.) all element greater than pivot is on RIGHT, side whereas

j, such that A[j] < pivot

A

10	5	8	12	15	6	3	9	16	∞

i, j

SWAP

A

10	5	8	9	15	6	12	16	∞

↓ continue same

A

A

10	5	8	9	3	6	15	12	16	∞

j, i

j become smaller than i stop.

all element on left is smaller than pivot.
hence PIVOT, reach it's correct place.

Pivot is SORTED

Perform QUICK SORT on each PARTITION

```
Algo QuickSort(l, h) {
    if (l < h) {
    j = partition(l, h)
    QuickSort(l, j)
    QuickSort(j+1, h)
    }
}
```

→ j is included to act like a ∞, as every number is smaller than this.

```
partition(l, h)
{ pivot = A[l]
  i = l, j = h
  while(i < j) {
    do {
      i++;
    } while(A[i] ≤ pivot)
    do {
      j--;
    } while(A[j] > pivot)
    if(i < j) {
      swap(A[i], A[j])
    }
  } SWAP(A[l], A[j])
  return j;
```

Quick Sort Analysis :-

Best case → O(n log n)

1. Best case is assumption partition is happening from median index.

2. $O(n^2)$ = when the list is sorted or almost sorted. To avoid → to select middle element as pivot & Random selection for pivot

2. → Stack size of System during Recursive call Worst O(n), Best O(log n).