DATA ANALYSIS AND ALGORITHM

- Algorithm

Finite set of steps to solve a problem.

→ characteristics of Algorithm

(i) Unambiguous

(There shouldn't be any ambiguity)
(It is always meaningful)
Eg: a⊥b x

: a = b+c; ✓

(ii) Finiteness

Finite number of steps to terminate

(iii) Input / output

At least 1 input, at max 1 output

(iv) Feasibility

→ Steps involved in solving a problem:-

(i) Identify problem statement.
(ii) Identify the constraints.
(iii) Design a logic
- → Divide & Conquer
- → Greedy Approach
- → Dynamic Prog
- → Branch & Bound
- → Backtracking

(iv) Validation (Algo works for every test case or not)

* Prove by PMI

* Prove by contradiction.

(v) Analyse (Time + Space)

→ Types of Analysis

Priory	Posterior
(i) Done before execution	(i) Done after execution

(ii) Frequency count of fundamental instruction
Eg:- int n;
int sum = 0;
cin >> n;
for(int i=0; i<n; i++)
sum += i;
	(ii)

(ii) we get to know estimated value	(ii) we get exact value
(iii) Uniform value $O(n^2)$, $O(n)$]→ will always be same	(iii) Dependent on the system inputs.
(iv) Does not depend on system.	(iv) Dependent on the system.
(v) Can be used to compare algorithms.	(v) Can't be used to compare algorithms.

→ ASYMPTOTIC NOTATIONS

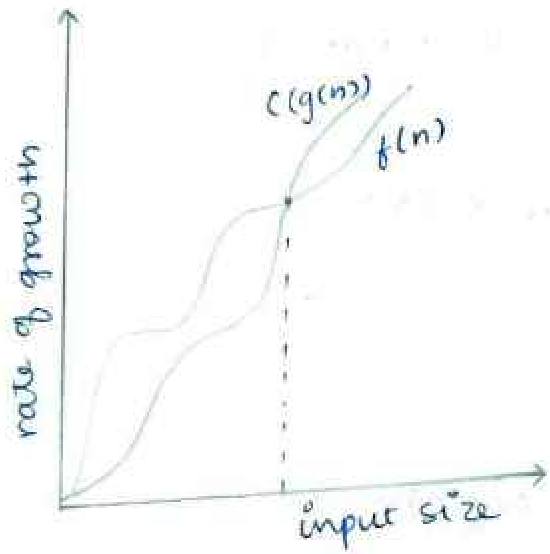
Used when input is large

(1) Big oh (O) (≥)

when we say,

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

g(n) is "tight" upper bound of f(n)



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

if $0 \le f(n) \le c(g(n))$ $\forall n \ge n_0$ for some constant

$c > 0$

<u>Rules</u>

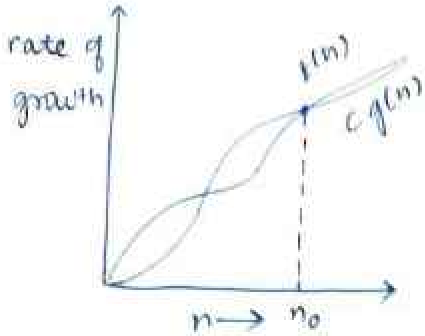
(i) constant are ~~ignored~~ ignored if it comes as:-

+, -, *, /

(ii) lower order term are ignored in +, -

(2) Big Omega (Ω) : $f(n) = \Omega g(n)$

if $f(n) \geq c(g(n)) \geq 0$ $\forall n \geq n_0$ and some constant $c > 0$



It is tight upper bound. (can also be equal)

(3) Small o : $f(n) = o(g(n))$

if $c(g(n)) > f(n)$ $\forall n > n_0$ ~~for~~ $\forall c > 0$

(similar graph)

It is upper bound.

(4) Small Omega (ω) : $f(n) = \omega g(n)$

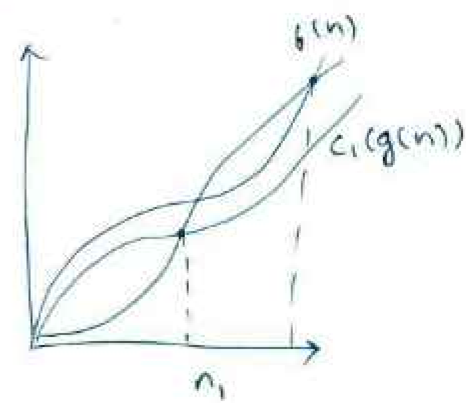
$\forall n > n_0$ & $\forall c > 0$.

if $cg(n) < f(n)$ $(cg(n) \leq 0)$

(similar graph)

(5) Theta Notation (θ) : $f(n) = \theta g(n)$

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



$\forall n \geq \max(n_1, n_2)$

& some constant c_1 & $c_2 > 0$

$f(n) = \theta(g(n))$ iff $f(n) = O(g(n))$ & $f(n) = \Omega(g(n))$

Note :-

1) $f(n) = O(g(n)) \rightarrow g(n) = \Omega f(n)$

2) $f(n) = o(g(n)) \rightarrow g(n) = \omega(f(n))$

$\boxed{cg(n) > f(n)}$ $\boxed{g(n) > f(n)}$

	(aRa) Reflexive	$\boxed{aRb \& bRa}$ Symmetric	aRb & bRc so, aRc Transitive
O	Yes	NO	Yes
o	NO	NO	Yes
Ω	Yes	NO	Yes
ω	NO	NO	Yes
θ	Yes	Yes	Yes

* Calculating Time complexities

(1) Single loop

(2) Nested loop

(3) if - else

(4) Recursive function

Single Loop

```
int sum = 0;        -> 1
for (int i = 0; i <= n; i++)    1, (n+1), (n)
    sum += i;       -> (n)
```

$= 1 + 1 + n + 1 + n + n$

$= 3n + 3$

Example:

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

/* $i = \underbrace{1, 3, 5, \ldots, n}_{x \text{ times}}$ (AP)

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

(x = no of times loop runs)

$n = 1 + 2x - 2$

$n = -1 + 2x$

$$\boxed{x = \frac{n+1}{2}}$$

Example:

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

/* $i = 1, 2, 4, 8, 16 \ldots, n$

nth term $= ar^{x-1} \geq n$

$2^{x-1} \geq n$

$\log_2(2^{x-1}) \geq (\log_2 n)$

$(x-1) \log_2 2 \geq \log_2 n$

$x - 1 \geq \log n$ */

Example:

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

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

(multiplication in nested loops)

Example

```
for (int i = n; i ≥ 1; i = 1/2)
{
    sum += i;
}
```

$i = n, n/2, n/4, n/8, \ldots 1$

$a = n, \; r = \frac{1}{2}$

$1 = n * \left(\frac{1}{2}\right)^{k-1}$

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

$= 2n = 2^k$ (take log both side)

Example

```
for (i=1; i≤n; i++)
{
    for (int j=1; j≤n; j++)
}                 O(1)
```

Sol

i	j	times
1	$1 \to n$	$(n+1)$
2	$1 \to n$	$(n+1)$
⋮	⋮	⋮
n	$1 \to n$	$(n+1)$
		$n(n+1)+1$

Example

```
for (i=1; i≤n; i++)
{ for (int j = 1; j≤n; j++)
}
  sqrt (i*j);
```

sqrt (i*j): $O(\sqrt{n})$ / $O(\log n)$ ✓

Example

```
for (int i = 1 ; i ≤ n ; i = i * 2)
  {
   for (int j = 1 ; j ≤ n ; j += 2)
   }
        O(1)
```

Time = $O(n \log n)$

i	j	times
1	$1 \to n$	$(n+1)$
2		$(n+1)$
4		$(n+1)$
8		⋮
⋮		⋮
n		$(n+1)$

$\log n$

$\log n * (n+1)$

Example

```
for (int i = 1 ; i ≤ n ; i++)
  {
   for (int j = 1 ; j ≤ n ; j += 2)
      {
        for (int k = 1 ; k ≤ n ; k *= 2)
      }
   }
      O(1)
```

i	j	k
1	1	$\log n$
2	3	$\log n$
3	5	⋮
⋮	7	⋮
⋮	⋮	
n	n	

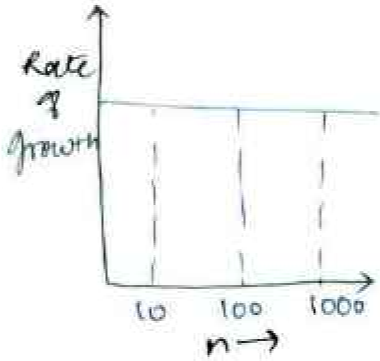
$n * \frac{n}{2} (\log n)$

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

* Types of complexities

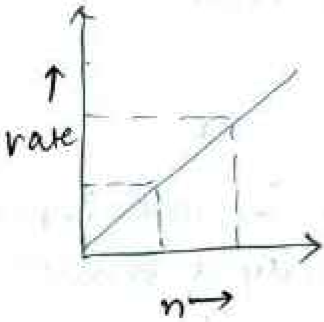
(i) Constant O(1) : Complexity is constant and doesn't change even after changing input size.

Eg:- Array - access an element.



(ii) linear O(n) : Complexity grows in direct proportion of input size.

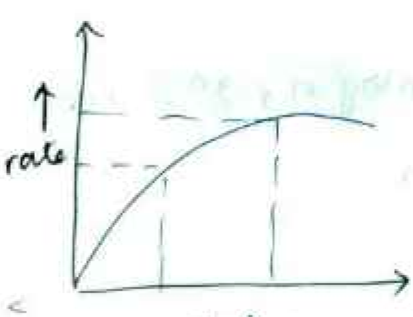
Eg:- Searching element in array.



(iii) logarithmic O(log n) : complexity grows linearly when input increases exponentially.

Eg:- Binary Search

~~Fon~~ for n inputs → x steps
2n inputs → x+1 steps



Ex:- 1024 → 10
512 → 9
512/2 → 8
|
|
|

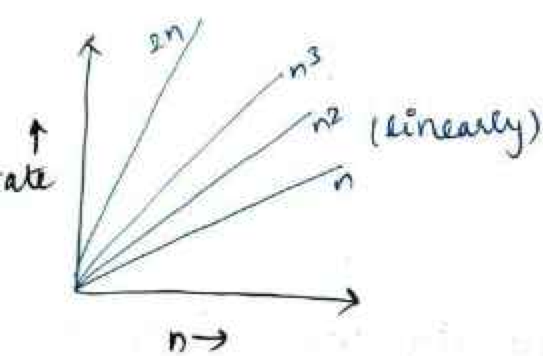
(iv) Polynomial (n^k): complexity grows in direct proportion of k^{th} power of input size.

k → constant

Eg:-

	n^3	n
$n = 10 \rightarrow$	10^3	10
$n = 100 \rightarrow$	10^6	100
$n = 1000 \rightarrow$	10^9	1000
$n = 10000 \rightarrow$	10^{12}	10000

(v) Exponential $O(k^n)$: complexity grows exponentially with linear increase in input size.
($2^n, 3^n, 4^n, 5^n, \ldots k^n$)



OR

with increase in one input size, complexity is multiplied by k.

Eg:- $2^{n+1} = 2^n \cdot 2$ (here $k = 2$)

ves

$n, \log n, \sqrt{n}, \log\log n, \log^2 n, \log(n!), n\log n, 2^n, n!$

→ Arrange in increasing order of operation.

ol

~~$\log\log n < \log^2 n < \log n < \log n! < n\log n < \sqrt{n} < n < 2^n$~~

~~$< n!$~~

Sol

$\log\log n < \log n < \log^2 n < \qquad < \sqrt{n} < n < \log n! <$

$n\log n$

★ Summation Method.

Eg:- ① for (int i = 1; i ≤ n; i++)

{ O(1)

}

$TC = \sum_{i=1}^{n} 1 = 1+1+1+1 \ldots + n \text{ times}$

$= O(n)$

Eg:- ② for (int i = 1; i ≤ n; i = i+2)

{ O(1)

}

$TC = \sum_{i=1,\, i=i+2}^{n} 1+1+1+1+ \ldots + \left(\frac{n+1}{2}\right) \text{ times}$

$= O(n)$

Eg:- ③ for (int i = 1; i ≤ n; i++)

{

O(n)

}

$TC = \sum_{i=1}^{n} n = n+n+n+ \ldots + n \text{ times}$

$TC = O(n^2)$

Eg:-

(a)
```
for(int i = 1; i <= n; i++)
    for (int j = i+1; j <= n; j++)
        O(1)
```

$$TC = \sum_{i=1}^{n} \sum_{j=i+1}^{n} 1$$

$$= \cancel{\sum_{i=1}^{n} ((n-1)+(n-2)+(n-3)+\dots+1)}$$

$$= \sum_{i=1}^{n} 1+1+1+1 \dots + (n-i) \text{ times.}$$

$$= \sum_{i=1}^{n} (n-i)$$

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

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

Ques

```
int i=1; s=1;
while (s<=n)
{
    i++;
    s=s+i;
}
```

$1 \quad 3, \quad 6 \quad 10 \quad 15 \quad \dots$

$2 \quad 3 \quad 4 \quad 5$

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

$$\boxed{K = \sqrt{n}}$$

(b) for (int i = 1; $i * i \le n$; i++)

$O(i)$

$k * k = n$

$\boxed{k = \sqrt{n}}$

(c) int j = 1; i = 0;

while (i<n)

{

i = i + j;

j++;

}

$i = 1, 3, 6, 10, \ldots$

$\boxed{k = \sqrt{n}}$

(d) for (i = 1; $i \le n$; i++) → n

{

for (j = 1; $j \le n$; j = j + i)

$O(1)$

}

i = 1

j = 1, 2, 3, 4, ... n

<u>n times</u>

i = 2

j = 1, 3, 5, 7 ... n

$4 \le \frac{k(k+1)}{2}$

$\frac{n}{2}$ times

i = 3

j = 1, 4, 7, 10, ...

$\frac{n}{3}$ times

for 2nd loop

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

$$= n\left[1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \cdots \frac{1}{n}\right]$$

$$\leq n\left[1 + \underbrace{\frac{1}{2} + \frac{1}{2}} + \underbrace{\frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4}} + \cdots\right]$$

$$\leq n\left[1 + 1 + 1 + \cdots\right]$$

$\leq n\,[\log n]$ terms

(e) for (int i = 1; i ≤ n; i++)
    for (int j = 1; j ≤ n; j++)
        O(1)

$O(n^2)$

(f) for (int i = 2; i ≤ n; i = pow(i, k))
        O(1)

★ Recurrence Relation

Recurrence Relation is an equation that defines a sequence based on a rule that given the next term as a func. of the previous term(s).

OR

A recurrence relation is used to determine the relation between the time complexity of the problem and time complexity of sub problem.

→ Binary Search

```
bool binary search (int * arr, int l, int r, int key)
{
    O(1) if (l > r)  return false;
    O(1) int mid = (l+r)/2          OR  l+(r-l)/2
    O(1) if (arr[mid] == key) return true;

        else if (arr[mid] < key)
             return binary search (arr, mid+1, r, key);

        else
         return binarySearch (arr, l, mid-1, key);
}
```

$T(n)$ — the three $O(1)$ lines; $T(n/2)$ — the recursive calls.

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

$T(1) = 1$

★ Solving RR

(i) Forward Substitution

Ex:-

$T(n) = T(n-1) + n$ // first $T(1)$ will be solved then $T(2)$ & so on ...

$T(1) = 1$

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

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

$\vdots$

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

$T(n) = 1+2+3+4+5+ \dots n$

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

$O(n^2)$

(ii) Backward Substitution

first $T(n)$ will be calculated, then $T(n-1)$ and so on --

(iii) Master Theorem.

Ques

Solve using backward substitution

$T(n) = T(n-1) + n$ — (i)

$T(1) = 1$

put $n = n-1$ in eq^n (i)

$T(n-1) = T(n-2) + n-1$ — (2)

$T(n) = T(n-2) + (n-1) + n$ — (3)

Put $n = n-2$ in eq^n (i)

~~T(n-1)~~

$T(n-2) = T(n-3) + (n-2)$ — (4)

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

for k^{th} term

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

$\Rightarrow T(n) = \boxed{T(n-k)} + (n-k+1) + \dots + n$

$\boxed{n-k=1}$ $\quad T(1)$

$\boxed{k = n-1}$

$\Rightarrow T(n) = T(n-(n-1)) + (\cancel{n}-(\cancel{n}-1)+1) + (\cancel{n}-(\cancel{n}-1)+2) + \dots + n$

$\Rightarrow T(1) + 2 + 3 + 4 + \dots + n$

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

Ques $T(n) = T\left(\frac{n}{2}\right) + 1$ —(i)

$T(1) = 1$

Put $n = \frac{n}{2}$ in eq^n (i)

$T\left(\frac{n}{2}\right) = T\left(\frac{n}{4}\right) + 1$ —(ii)

$T(n) = T\left(\frac{n}{8}\right) + 1 + 1$ —(iii)

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

~~$T(n) = T\left(\frac{n}{16}\right) + 3$ —(iv)~~

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

$T(n) = T\left(\frac{n}{8}\right) + 1 + 1 + 1$ —(v)

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

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

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

$$n = 2^k$$

$$\boxed{k = \log n}$$

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

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

22/03/22

→ MASTER's THEOREM

$T(n) = aT(n/b) + f(n)$

3 cases:-

case ① $f(n) = O(n^{\log_b a - \epsilon})$

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

case ② $f(n) = \Theta(n^{\log_b a} \log^k n)$ (mostly at $k=0$)

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

case ③ $f(n) = \Omega(n^{\log_b + \epsilon})$

$T(n)$ ~~$f(n)$~~ $: \theta(f(n))$

and $f(n)$ follows regularity condition $(af(n/b) \leq cf(n)$ where $c<1$

then $T(n) = f(n)$

Example :-

① $T(n) = T(n/2) + n$ ③ $= T(n) =$

② $T(n) = 2T(n/2) + n$ ②

③

④ $T(n) = 3T(n/2) + n^2$

Sol $a = 3, b = 2, f(n) = n^2$

find $n^{\log_b a} = n^{\log_2 3} \Rightarrow n^{1.5 \ldots}$

$n^2 = O(n^{1.5})$

⑤ $T(n) = 4T(n/2) + n^2$

$a = 4,\ b = 2,\ f(n) = n^2$

$n^{\log_b a} = n^{\log_2 4} = n^2$

Case 1 :- $n^2 = O(n^{2-\epsilon})$ (Invalid)

Case 2 :- $n^2 = \Theta(n^2 \log^k n)$ for $k = 0$

$\boxed{\text{So, } T(n) = \Theta(n^2 \log n)}$

⑥ $T(n) = 16T(n/4) + n$

$a = 16,\ b = 4,\ f(n) = n$

$n^{\log_b a} = n^{\log_4 16} = n^2$

Case 1 :- $n = O(n^{2-\epsilon})$ valid

This f^n will always be greater for any small constant value of ϵ.

$\boxed{\text{So, } T(n) = O(n^2)}$

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

master's theorem cannot be applied as a & b ~~ca~~ is not a constant.

a & b should be constant, where $a \geq 1$, $b \geq 1$

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

$a = 2$, $b = 4$, $f(n) = n^{0.51}$

$n^{\log_b a} = n^{\log_4 2} = n^{0.5}$

<u>Case 1</u> :-

$n^{0.51} = O(n^{0.5-\epsilon})$ <u>Invalid</u>

<u>Case 2</u> :-

$n^{0.51} = \Theta(n^{0.5} \log^k n)$ (where $k \geq 0$) (Invalid

<u>Case 3</u> :- $n^{0.51} = \Omega(n^{0.5+\epsilon})$ (valid)

$\hookrightarrow$ v.v.v.v. small value

Regularity condition for case 3.

$$\underbrace{2 * \cancel{f}\left(\frac{n}{4}\right)^{0.51}} \leq c * n^{0.51}$$

almost cancels out but this value will be a bit smaller than 1. ∴ Regularity condition is valid.

Practice

① $T(n) = 4T(n/2) + \log n$ case 1 $(O(n^2))$

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

③ $T(n) = 3T(n/3) + n/2$ case 2 $(O(n\log n))$

④ $T(n) = \sqrt{2}\,T(n/2) + \log n$

⑤ $T(n) = 0.5\,T(n/2) + 1/n \rightarrow$ cant apply bcz $a \geq 1$

⑥ $T(n) = 64\,T(n/8) - n^2 \log n \rightarrow$ master's theorem cannot be applied bcz '−'

Sol 1

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

$a = 4, \; b = 2, \; n = \log n$

$n^{\log_b a} = n^{\log_2 4} = n^2$

Case 1 :-

$\log n = O(n^{2-\epsilon})$ (valid)

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

Sol 2

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

$a = 7, \; b = 3, \; n = n^2$

$n^{\log_b a} = n^{\log_3 7} = 0$ $\quad (1 < n^{\overset{1.5}{\log_3 7}} < 2)$

Case 1 :-

$n^2 = O(n^{\log_3 7 - \epsilon})$ Invalid

Case 2 :-

$n^2 = \Theta(n^{\log_3 7} \log^k n)$ Invalid

Case 3 :-

$n^2 = \Omega(n^{\log_3 7 + \epsilon}) \rightarrow$ valid

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

Regularity condition.

$$\underbrace{7 * \left(\frac{n}{9}\right)^2}_{\text{value will very small}} \leq c * n^2$$

value will very small if $c = 0.99\ldots$ (valid)

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

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

$a = 3, \; b = 3, \; n = n/2$

$n^{\log_b a} = n^{\log_3 3} = n^1$

Case 1 :-

$n/2 = O(n^{1-\epsilon})$ Invalid

Case 2 :-

$n/2 = \Theta(n^1 \log^k n)$ valid for $k = 0$.

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

Sol 4 $T(n) = \sqrt{2}\, T(n/2) + \log n$

$a = \sqrt{2},\ b = 2,\ n = \log n$

$n^{\log_b a} = n^{\log_2 \sqrt{2}} = n^{0.5}$

Case 1

$\log n = O(n^{0.5-\epsilon})$ valid

$\boxed{T(n) = O(n^{1/2})}$

Ques
Imp

```
T(n)
{
  if (n ≤ 1) return 1;
  else return T(√n);
}
```

write recurrence relation for the given code & solve it using master's therorem

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

This recurrence relation is in a form where MT can't be applied but we can convert it in substitution form.

Suppose $\boxed{n = 2^m} \Rightarrow \boxed{m = \log n}$

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

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

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

Suppose $S(m) = T(2^m)$

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

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

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

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

Case 1

$1 = O(m^{0-\epsilon})$ Invalid.

Case 3

$1 = \Omega(m^{0+\epsilon})$ Invalid.

Case 2

$1 = \Theta(m^0 \log^k m)$

valid for $k = 0$

$1 = \Theta(1)$

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

$\boxed{m = \log n}$

$S(m) = \Theta(\log \log n)$

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

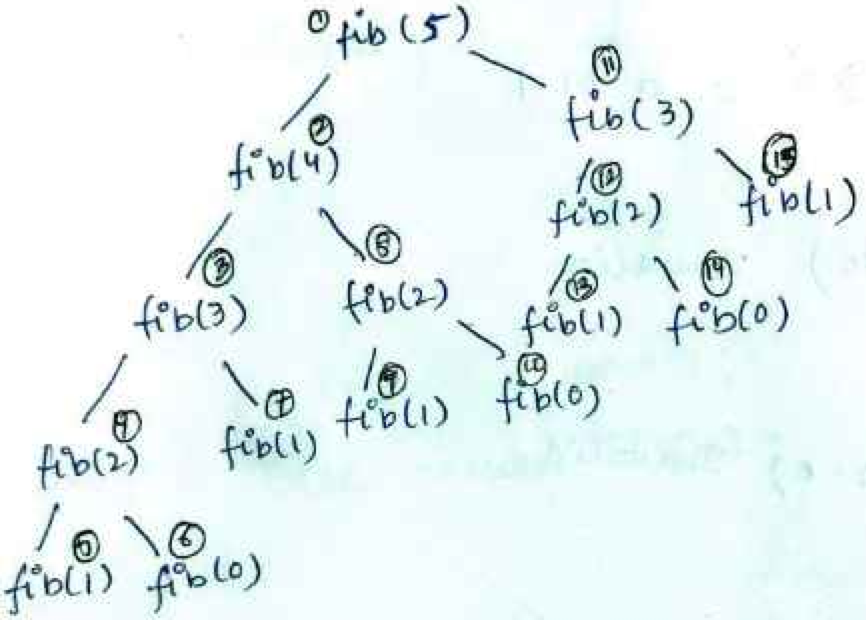
23 March 2

Fibonacci Series :-

0, 1, 1, 2, 3, 5, 8, 13, ...
(0) (1) (2) (3) (4) (5) (6) (7)

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

$fibo(4) = fibo(3) + fibo(2)$
$= 2 + 1 = 3$




```
fib(n)
{ if (n ≤ 1) return n;          — O(1)
  return fib(n-1) + fib(n-2);
}          (T(n-1))    (T(n-2))
```

$T(n) = T(n-1) + T(n-2) + 1$ — ①

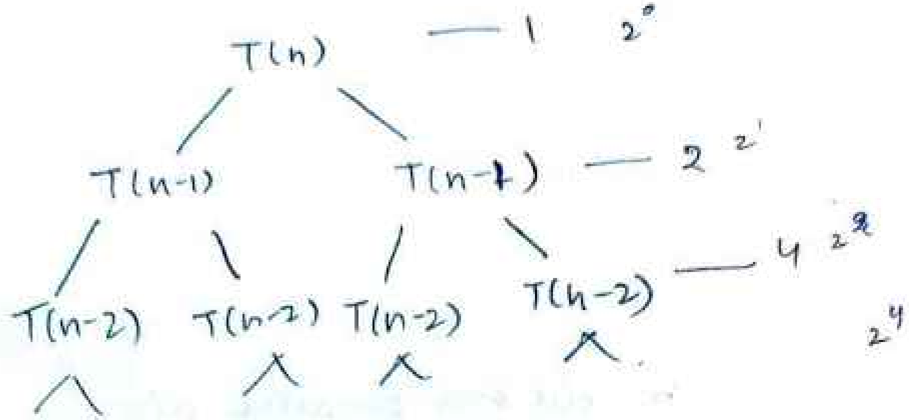
$T(n) = T(n-1) + T(n-1) + 1$ — ②

eqⁿ ② > eqⁿ ①

$T(n) = T(n-2) + T(n-2) + 1$ — ③

Eqⁿ ③ > Eqⁿ ②

Tree Method



$T(1)\ T(1)\ T(1) - - - -\ T(1) - - -\ 2^{n-1}$

$T(n) = 1 + 2 + 4 + 8 + - - - + 2^{n-1}$

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

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

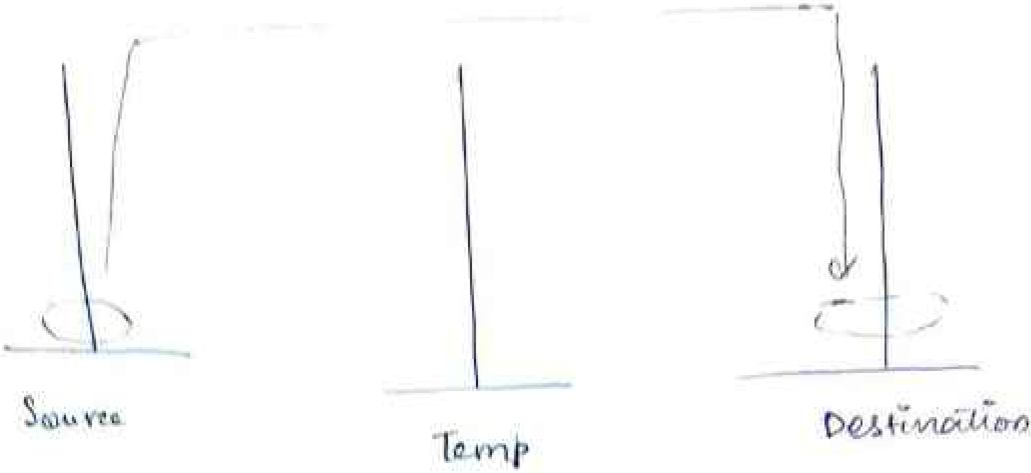
Similarly, solve for $T(n) = 2T(n-2) + 1$

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

Itrative

```
fib(n)
{
   if (n≤1) return n;
   int a=0, b=1, c;
   for( i=2 to n)
   {
     a
```

* Tower of Hanoi:



Rules:

1. large disc can't be put over smaller disc.
2. You can't move more than 1 disc at a time.

Steps for 2 discs:

1. move 1 disc from A to B using C.
2. move 1 disc from A to C.
3. move 1 disc from B to C using A.

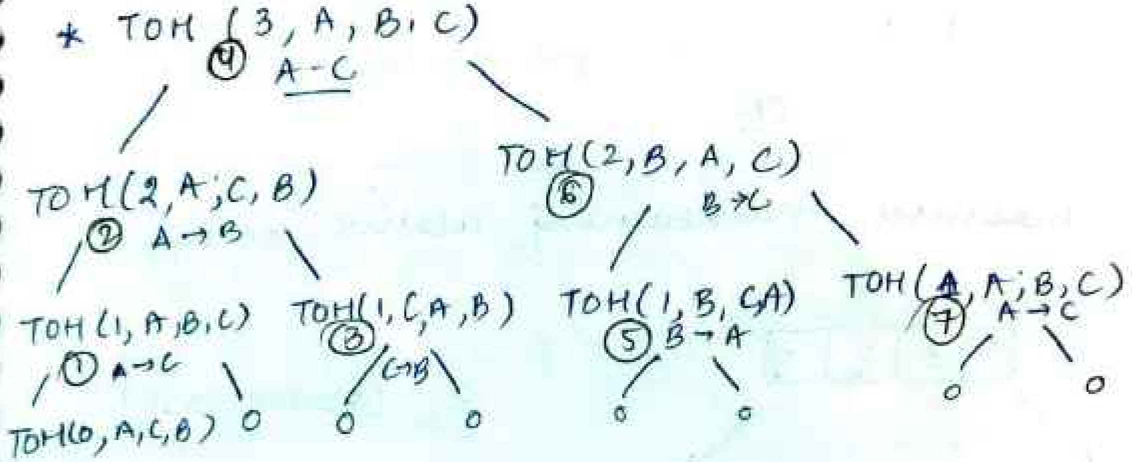
Steps for 3 discs:

1. move 2 discs from A to B using C.
2. move 1 disc from A to C.
3. Move 2 discs from B to C using A.

Steps for n discs:

1. Move (n-1) discs from A to B using C.
2. Move 1 disc from A to C.
3. Move (n-1) discs from B to C using A.

```
void TOH (int n, int A, int B, int C)
{
    if (n>0)
    {
      TOH(n-1, A, C, B);
      print(" move a disc from A to C");
      TOH(n-1, B, A, C);
    }
}
```



$T(n) = 2T$

28th Mar '22

UNIT-2

① Inplace Sorting Algo:- The sorting algorithm that does not take extra space.

Ex:- Bubble, insertion, selection.

② Stable Sorting Algo:- Relative order of element does not change.

③ Unstable Sorting Algo:- Relative order of elements might change after sorting.

OR

It doesn't guarantee maintaining relative order.

Ex:-

4	3	1_A	1_B

(~~Stable sort~~)

1_A	1_B	3	4

(* Stable sort)

1_B	1_A	3	4

(* unstable sort)

Ex:- Bubble, insertion, selection

Eg:- Unsorted array:-

3	1	2_A	2_B	4

↓

Sorted array:-

1	2_A	2_B	3	4

(There are chances that unstable algo give this result)

what can you say about this sorting algo?

① It is stable sort.
② It is unstable sort.
③ It might be stable. ✓
④ It might be unstable ✓

④ Online Sorting Algo:- This kind of sorting algo doesn't need whole array in the beginning of execution.

It can process elements one by one as elements appears.

Eg:-

2	3	1	0	9	8	6	2

• Insertion is online sort algo.

Ques Find min/max in a series of elements.

• Compare and change max/min at every step.

⑤ Internal and External sorting Algo:- Need whole array in RAM (physical memory) ~~or~~ during execution. (internal)

Part of the array reside in RAM during execution (external)

Eg:- You are given an array of 10GB, that needed to sort using a RAM of 1GB. How you are going to do that?

Sol

							...

k-parts.

Put a chunk in memory, sort that chunk,

Put it back to HDD (virtual memory)

Merge all parts to make final sorted array.

Sorting concept - Merge Sort.

Bubble Sort

for ~~a~~ $i=1$ to $n-1$ – Pass

  for $j=1$ to $n-i$

    if $(a[j] > a[j+1])$

      swap $(a[j], a[j+1])$

$i=1,\ n-1$
$i=2,\ n-2$
$i=3,\ n-3$
⋮
$i=n-1,\ n-n-1=1$

* Swap i^{th} & j^{th} element if $a[i]>a[j]$ for $i<j$.

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

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

space - $O(1)$

* After i^{th} iterations, i elements reach their correct positions.

Ex:-

4	3	1_A	2	1_B

1^{st} Pass

3	1_A	2	1_B	4

2^{nd} Pass

1_A	2	1_B	3	4

3^{rd} Pass

1_A	1_B	2	3	4

* Bubble sort is stable sort.

* Loop Invariant condition :- A loop invariant is a condition that is necessarily true immediately before and after each iteration of a loop.

→ LIC for Bubble Sort - after i^{th} iteration i elements will reach their correct positions.

Eg:-

54	26	93	17	77	31	44	55	20

what will be the status of array element after 4^{th} Pass?

1^{st} Pass:-

26	54	17	77	31	44	55	20	93

2^{nd} Pass:-

26	17	77	31	44	55	20	77	93

3^{rd} Pass:-

17	26	31	44	54	20	55	77	93

4th Pass :- | 17 | 26 | 31 | 44 | 20 | 54 | 55 | 77 | 93 |

Q what will be the status of array after 8th pass?

Elements will be sorted after 8th pass.

Number of Comparisons in bubble Sort :- $O(n^2)$

* Swapping depends upon the array elements:-

Sorted array: Best case

Reversed Sorted Array:- Worst case

1, 2, 3, 4, 5 → No swapping

5, 4, 3, 2, 1 → 4+3+2+1 - swaps

Insertion Sort : {Shifting Based Algo}

(Increment Sort)

```
for j=2 to n
  key = a[j]
  i=j-1
  while i>0 & a[i]>key
     a[i+1] = a[i]
       i = i-1
     a[i+1] = key
```

Ex :-

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

i=1 j=2 key=7

→

6 7

2	7	6	1	3	4

Loop Invarient : first i elements are sorted after ith pass.

① Best Case :- Sorted array

→ loop will not get executed and it will keep on changing key with itself

{Best for Nearly or Partially Sorted}

→ $O(n)$

Worst case :- $O(n^2)$ (reversed sorted array)

5	4	3	2	1

Ques

-1	4	7	2	3	8	-5

what will be array after 4th pass?

$i = 1, j = i+1$ key = 4

I st pass

1	2	4	7	3	8	-5

II nd pass

-1	2	4	7	3	8	-5

III rd

-1	2	3	4	7	8	-5

Selection Sort :-

```
for j=1 to n-1
  min = j
  for i = j+1 to n
    if (a[i] < a[min])
      min = i
    swap(a[j], a[min])
```

→ This algo works for finding min element at each pass & then placing it at its correct position in the array.

Note :- 1 swapping for each pass, max swapping :- $O(n)$

Eg :-

1	2	3	4	5

Sorted array comparisons but swapping to itself.

5	4	3	2	1

→ write another version of algo which finds max element and swap after 1st pass.

Ques.

-1	5	3	9	12	4	8	23	15

→ Sort this array using selection sort and find the number of comparisions and swapping.

$\frac{9 \times 8}{2} = 36$ ← comparisions

swapping → 4

Quick, merge, Heap, Insertion, selection, bubble ⇒ Comparision Based Algo.

Note:- No. of comparision on selection sort ⇒ $\frac{n(n-1)}{2}$

Note:- Tightest upperbound that represents no. of swaps required to sort using selection sort is – $O(n)$.

- Bubble, Selection, Insertion → Inplace sorting.
  - Bubble ↓ Stable
  - Selection ↓ unstable
  - Insertion ↓ Stable.

eg

5	4	2	1_A	1_B

↓

1_A	1_B	2	4	5

9	1_A	5	2	1_B

↓

1_A	1_B	2	4	5

This example gives stable results, however the selection sort is unstable.

Counting Sort

Counting is not comparision based.

Assumption:- Numbers to be sorted are in range {0, 1, 2 .. K}

Input:- A[1...n], where A[j] ∈ [0, 1, ... K] for j = 1, 2, ... n.

Output:- B[1 ... n], sorted B is assumpted to be already allocated and is given as a parameter.

Auxiliary storage:- C [0 ... K]   K → highest element.

Ex:- $n=5$

8	6	7	1	2

0 – 8

* counting takes extra space
$O(K+n)$

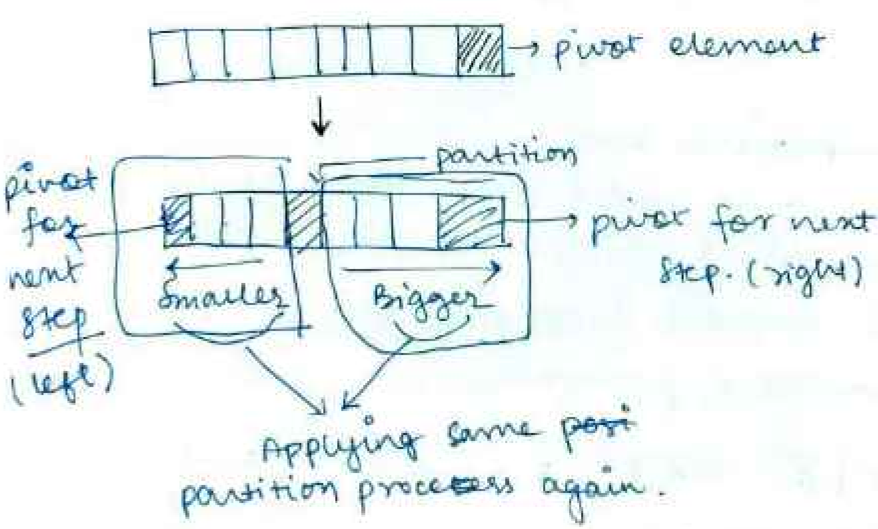
* Good for elements in the range (0-K) when K is at max n.
  → which means counting sort is going to take a lot of extra space if $K >> n$.

Quick Sort

→ Divide and conquer stratergy.

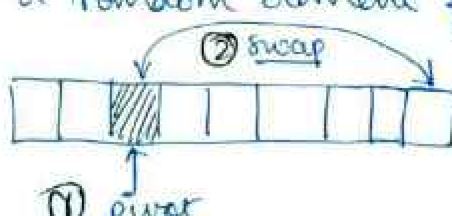
→ we pick a pivot(x) element and partition the array with respect to x, whose elements smaller than x are in one partition and greater elements are on another partition.

* which element can be chosen as pivot?
  → Any element, but here we will take right most element as pivot in each step.



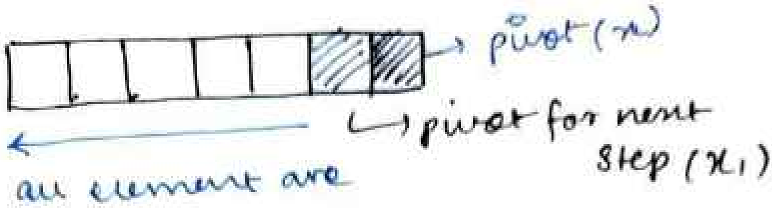
→ Randomised Quick Sort

- we take a random element for pivot



First we pick a random pivot & swap it with last element.

Worst case → already sorted array OR reversed sorted array.



all element are already $\leq x$

elements $\leq x_1$

Quick Sort Algo (A, P, r)

A[q] is already in its correct position.

```
if (P<r)
    q = Partition(A,P,r);
    QuickSort (A, P, q-1);
    QuickSort (A, q+1, r);

Partition (A, P, r)
    x = A[r]  //pivot
    i = P-1
    for (j=P to r-1)
        if (A[j] ≤ x)
        i = i+1
        swap(A[i], A[j])
    swap (A[i+1], A[r])
    return i+1;
```

Example

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

← x (pivot)

$p = 1, \ r = 6, \ x = 5$

$i = p - 1 = 1 - 1 = 0$

QuickSort (A, 1, 6)

q = Partition (A, 1, 6)

Step ① j = 1, 2, 3, 4, 5

if (A[1] ≤ 5)

i = 1

Swap(A[1], A[1])

Step ② j = 2, A[2] ≤ 5 (false)
(8)

Step ③ j = 3, A[3] ≤ 5 (false)

Step ④ j = 4, A[4] ≤ 5 (true)

i = 2

Swap (A[2], A[4])

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

⇒

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

Step ⑤ j = 5, A[5] ≤ 5 (true)

i = 3

Swap (A[3], A[5])

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

⇒

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

Swap (A[i+1], A[r])

↳ Swap (A[4], A[6])

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

⇒

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

return 4;

QuickSort (A, 1, 3)

QuickSort (A, 5, 6)

Quicksort (A, 1, 3)

q = Partition (A, 1, 3)

step ① = j = 1, 2

if (A[1] ≤ 3)

i = 1

swap (A[1], A[1]) (self swap)

step ② j = 2

A[2] ≤ 3 , i = 2

swap (A[2], A[2]) (self swap)

step ③ swap(A[i+1], A[r])

swap (A[3], A[3]) (self swap)

Next call

Quicksort (A, 1, 2)

step ① = j = 1

~~if (A[1] ≤ 1~~

swap (A[1], A[2])

→ Divide and conquer :- Break the problems into subproblems of same type. (Recursively solve these problems)

Combine : Combine the solution of smaller problems.

Ex: Binary Search, Quick Sort, Merge Sort.

→ Complexity Analysis :-

① $T(n) = T(k) + T(n-k+1) + n$

② $T(n) = T(n/2) + T(n/2) + n$ [Best Case]

③ $T(n) = T(n-1) + n$ [Worst Case]

④ $T(n) = T(n/10) + T(9n/10) + n$ [90 : 10]

* **Worst case**

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

$T(1) = 1$

* Quick Sort is $O(n^2)$ in worst case, still is considered best for practical uses. why?

Tree method

$T(n) \rightarrow n$

$T(n-1) \rightarrow n-1$

$T(n-2) \rightarrow n-2$

⋮

$T(1) \rightarrow 1$

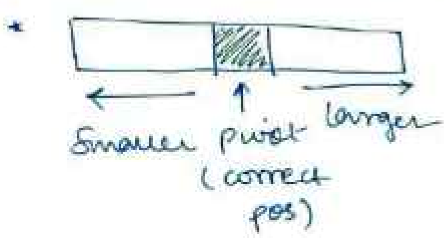
$T(1) = T(0) + 1$

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

$\rightarrow 1 + 2 + 3 + \dots + (n-2) + (n-1) + n$ (sum of n natural num)

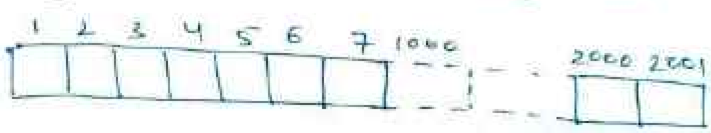
$\boxed{O(n^2)}$

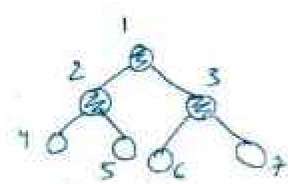
Ans :- Using Randomized Quick Sort we can achieve $O(n \log n)$ almost all the times.



* locality of reference (OS)

* **Comparing Quick Sort with Heap Sort.**





Present at index i

$\rightarrow$ children at $2i$ & $2i+1$

Page Fault :- No data ~~fond~~ found

Heap & Merge (Best, avg, worst) $\rightarrow O(n \log n)$

Bubble, Selection, Insertion $\rightarrow O(n^2)$

Bubble ↓ Swapping

Selection ↓ Comparison

Insertion ↓ Nearly sorted array

* **Quick Sort Best Case**

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

Masters theorem

$n^{\log_b a} \rightarrow n^{\log_2 2} = n$

Case 2:-

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

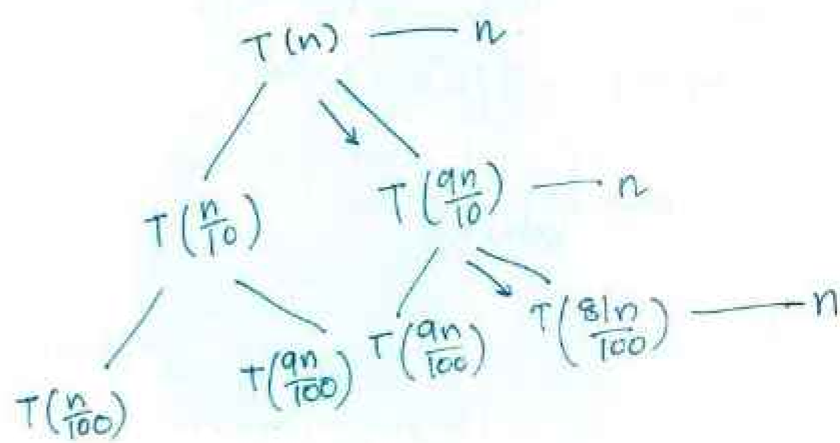
then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

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

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

* Array is divided in 90% & 10% ratio at each step. What will be the time complexity in this case. What will be the difference in heights of two ~~express~~ extremes of recursion tree?

$T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + n.$



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

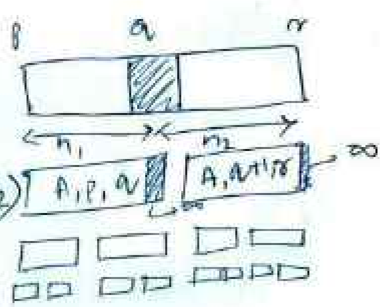
Difference:- $(\log_{10} n - \log_{10/9} n)$

* Stable — NO (relative order)
* Inplace - Takes space in recursive calls but not in
* ~~Online~~ - input manipulation so can be called inplace using its broad definition of using extra space.
* Online - NO (will require the entire array for choosing pivot)

→ **Merge Sort** :- Divide & Conquer

- Divide :- the element sequence to be sorted into subsequence of n/2 elements each.
- Conquer - Sort two subsequences recursively using merge sort.
- Combine - Two sorted subsequence to produce the sorted array.

```
MergeSort (A, P, r)        T(n)
  if (P<r)
  q = [(P+r)/2]
  MergeSort (A, P, q)      T(n/2)
  MergeSort (A, q+1, r)    T(n/2)
  Merge(A, P, q, r)
```




```
Merge(A, P, q, r)
n1 = q - P + 1
n2 = r - q
Let L = [1, 2, .... n1+1]
  R = [1, 2, ... n2+1]
  for (i=1 to n1)
   L[i] = A[P+i-1]
   for (j=1 to n)
```

$R[j] = A[q + j]$

$L[n_1 + 1] = \infty$

$R[n_2 + 1] = \infty$

$i = 1, j = 1$

for $K = p$ to r ← left subpart

  if $L[i] \leq R[j]$ → right subpart

    $A[K] = L[i]$

    $i = i + 1$

  else

    $A[K] = R[j]$

    $j = j + 1$

$i = 1,2,3$ L | 4_a | 9 | 11 |

$j = 1,2,3$ R | 4_b | 6 | 10 | ∞ |

4_a	4_b	6	9	10	11
1	2	3	4	5	6

$p = 1, r = 6$

while($i < n_1$)

{

  $A[K] = L[i]$

  $i{+}{+}$;

  $k{+}{+}$;

}

while ($j < n_2$)

{

  $A[K] = R[j]$

  $j{+}{+}$;

  $K{+}{+}$;

}

Time complexity

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

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

Space - $O(n)$

Stable - Yes.

Online - No

Example

1	2	3	4	5	6
1	4	2	8	-1	0

→ MergeSort (A, 1, 6) → MergeSort (A, 1, 3)
              ↓ ↓   → Merge (A, 4, 6)
              p r

$p < r$ (true)

$q = \left\lfloor \frac{1+6}{2} \right\rfloor = 3$

L

R

→ MergeSort (A, 1, 3)

q = 2

MergeSort (A, 1, 2) — p < r true

MergeS(A, 3, 3) — p < r false

q = 1

MS (A, 1, 1)   MS (A, 2, 2)

Merge

L | 1 | ∞ | R | 2 | ∞ |

| 1 | 4 | ∞ |

Inversion Count using Merge Sort

- unsorted | 2 | 1 | 3 | 7 |
- sorted | 1 | 2 | 3 | 7 |

for $i < j$

$A[i] > A[j]$

* 2 1 3 7
  1 2 3 7

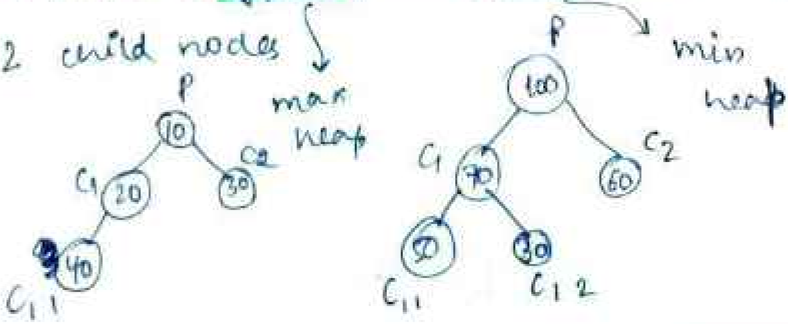
Inversion - 1

(No. of intersection = No. of inversions)

→ Heap Sort

Heap Sort uses a data structure called heap (Binary Heap).

- Binary Heap:- Binary heap is a complete binary tree where items are stored in such a way that prese parent node is greater or smaller than values in its 2 child nodes



Binary heap can be either max or min heap.

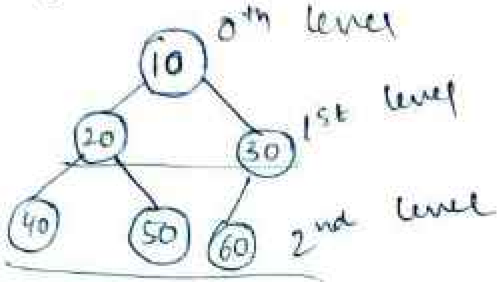
* Application of Binary Heap:-

① Heap Sort

② Graph Algorithm (Priority Queue)

- Dijkstra's shortest path
- Prim's Algo (Spanning tree)

③ k^{th} largest or k^{th} smallest number.



Smallest no. :- 0 level
2nd smallest :- 1 level
3rd smallest :- 1 level
4th, 5th, 6th, 7th :- 2nd level

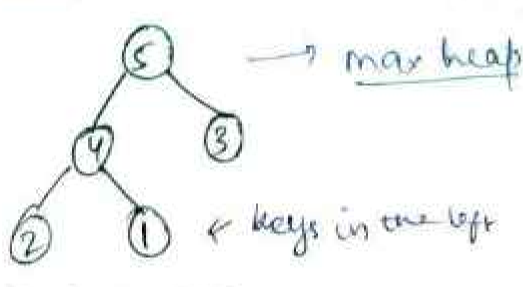
- To find 3rd Smallest

extract ~~find~~ - main() - Run this function 3 times.
& heapify

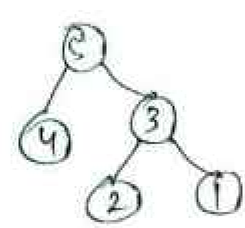
④ Merge k-sorted arrays

* Complete Binary tree :- A Binary tree is a complete Binary tree if all the levels are completely filled except possibly the last level and the last level has all the keys as left as possible.

Ex:-



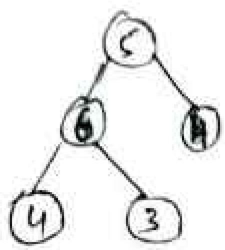
It is a CBT



Not a CBT

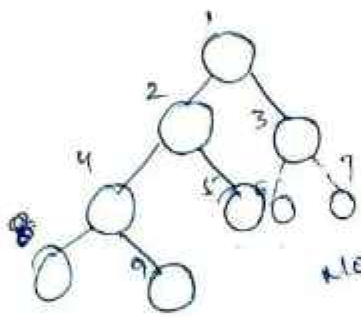
Not a heap

* Every heap (max / min) is a CBT but vice versa is not true.



CBT ✓
Heap ✗

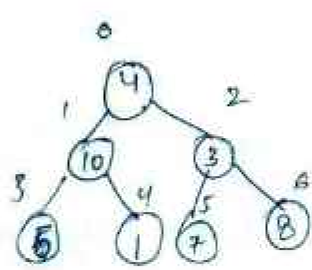
Ex:- How to check if a tree is CBT?



Node 6,7 are missing not a CBT.

Ex:-

0	1	2	3	4	5	6
4	10	3	5	1	7	8



Not a heap.
we need to use heapify to convert it into a max / min heap.

Parent i^{th} index

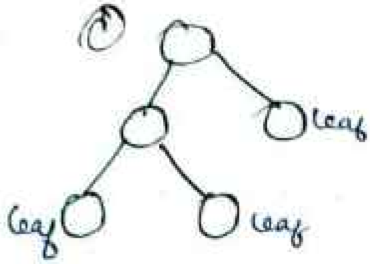
left child at $- 2i+1$
right child at $- 2i+2$ } for index beginning with 0

$(2i, 2i+1)$

→ **Full Binary Tree (FBT)**

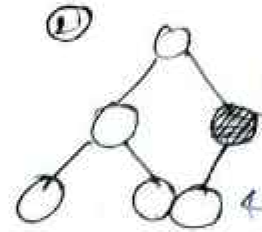
A binary tree is a full binary tree if every node has 0 or 2 children. We can also say that all nodes except leaf nodes have two children.

Ex :- ①



FBT - Yes
CBT - Yes
PBT - NO

②



FBT - NO
CBT - Yes
PBT - NO
only 1 child

→ **Perfect Binary Tree (PBT)** :- All internal nodes have two children & all leaf nodes are at the same level.

Que which of the following statement are true?

① All CBT are FBT → False
② All FBT are CBT → false
③ All PBT are CBT and FBT —
④ All CBT are min/max heap - False
⑤ All heaps are CBT. - True
⑥ All CBT are PBT - False
⑦ All FBT are PBT - False

* Perfect Binary tree is a balanced Binary tree

→ Height of PBT is $(\log_2 n)$ where n is the no. of nodes in tree.

→ No. of leaf nodes in PBT = No. of internal nodes + 1

2 edges → height = 2

No. of nodes in a PBT of height $h = 2^{(h+1)} - 1$

→ **Balanced Binary Tree**:- If height is $O(\log n)$

Ex:- AVL $|H_L - H_R| = 0$ or 1

Red Black Tree - No. of black nodes from every root to leaf paths is same and there is no adjacent red node.

* BBT are performance-wise very good as they provide $O(\log n)$ time for search, insert & delete.

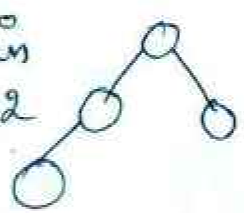
<u>Ques 1</u> what are min & max no. of elements in a heap of height h?

<u>Ques 2</u> Is an array that is in reverse sorted order a heap?

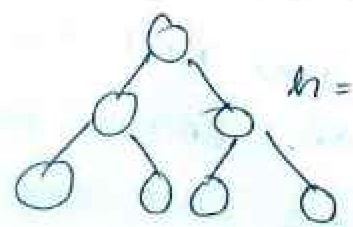
<u>Ques 3</u> is the sequence < 23, 17, 14, 6, 13, 10, 1, 5, 7, 12 > a heap?

<u>Sol 1</u>

min $h=2$



max $h=2$

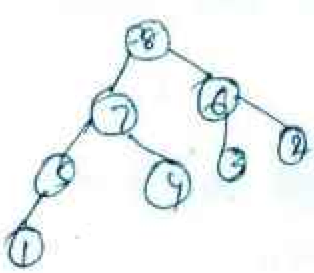


min - 2^h

max - $2^{h+1} - 1$

<u>Sol 2</u>

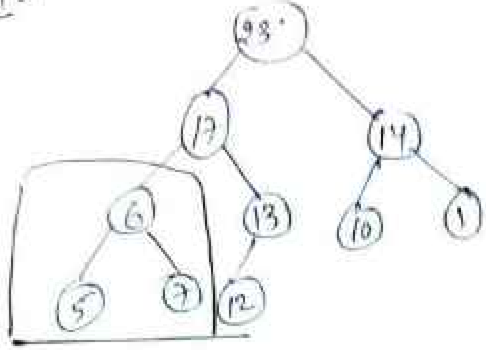
8, 7, 6, 5, 4, 3, 2, 1



Yes, max heap.

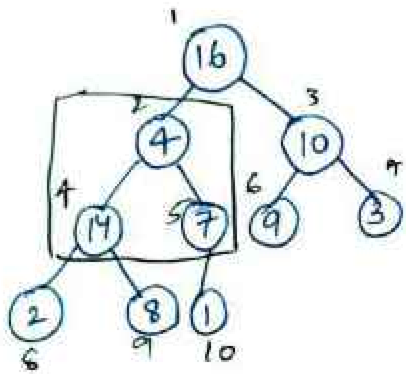
Sol 3

Sol 3



violating property of max heap.

* Heapify (Maintaining the heap property)



The property of max-heap is violated a index 2, so we can use heapify function at index 2.

* The function of heapify is to float down the value of index 2 which is violating the property of max heap.

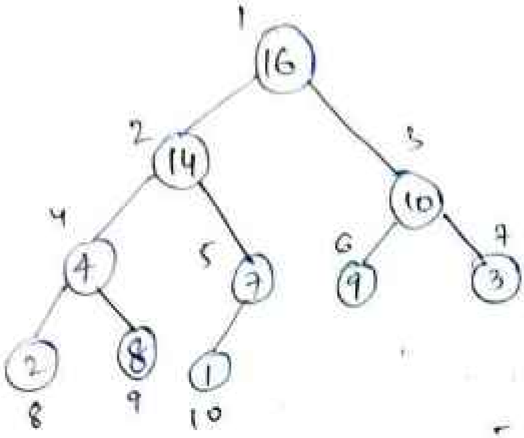
- heapify function.

```
→ Heapify (A, i)
      l = left (i) // 2i (index of left child)
      r = right (i) // 2i+1 (index of right child)
    if l ≤ heap.size (A) and A [l] > A[i]
      largest = l
    else largest = i
    if r ≤ heap.size (A) and A[r] > A [largest]
        largest = r

    if largest ≠ i
      swap(A[i], A [largest])
      Heapify (A, largest)  ( T= 2n/3)
```

Example

(1)



This is after 1st step of heapify.

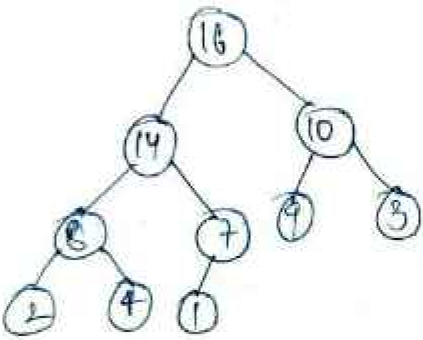
(index)

$i = 2, l = 4, r = 5$

largest = 4

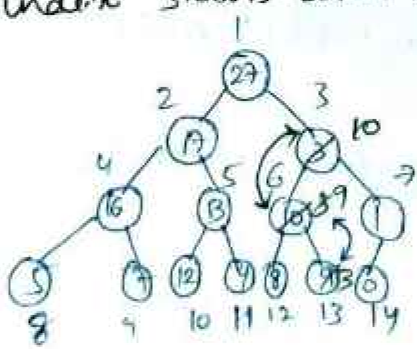
- Apply heapify on index 4

(2)



Example, A = 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0

Illustrate the operation of heapify (A, 3) on array A. where index starts with 1.



(index)

$i = 3$

$l = 6, r = 7$

largest = 6

swap(A[3], A[6])

heapify(A, 6)

$i = 6$

$l = 12, r = 13$

largest = 13

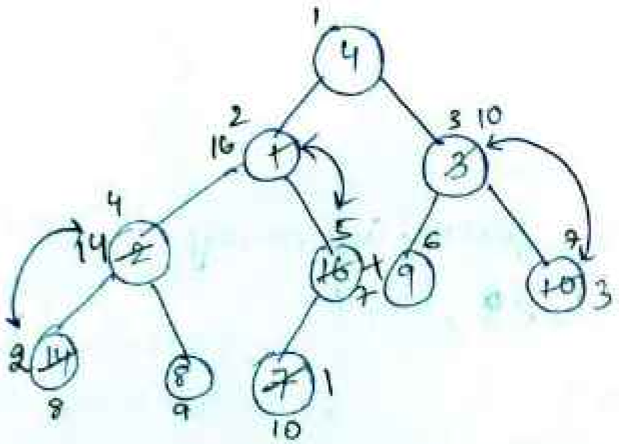
swap(A[6], A[13])

* BuildHeap Function

Buildheap (A) $\rightarrow$ O(1)
heapsize (A) = length (A)
for i = [length (A)/2] to 1
heapify (A, i) $\rightarrow$ O(log n)

→ why buildheap starts/begins heapify from [length (A/2)] ?

Ex:- 4, 1, 3, 2, 16, 9, 10, 14, 8, 7



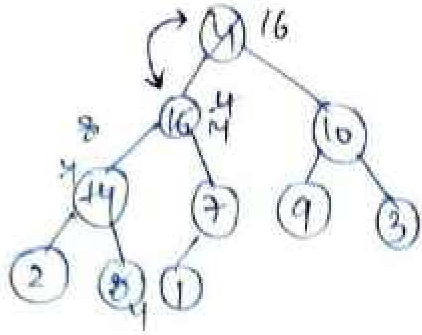
Almost half nodes are at leaves so there is no benefit of applying heapify at leaf nodes

n = [length (A)/2] = 5
So, heapify will begin at 5.
→ No change with Heapify (A, 5)

(1) Heapify (A, 4)
swap (A[4], A[8]), heapify (A, 8)

(3) Heapify (A, 3)
Swap (A[3], A[7]), heapify (A, 7)

(4) Heapify (A, 2)
Swap (A[2], A[5]), heapify (A, 5)
swap (A[6], A[10]), heapify (A, 10)



Heapify(A,1)

Swap (A[1], A[2])

Heapify (A, 2)

swap (A[2] , A[4])

heapify (A, 4)

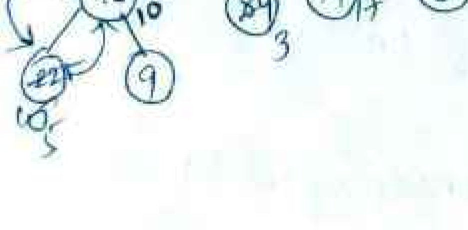
Swap (A[4] , A[9])

Heapify (A,9)

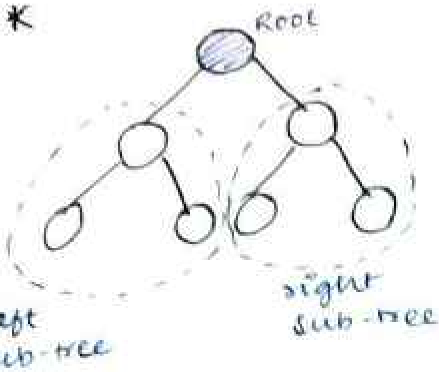
Ques

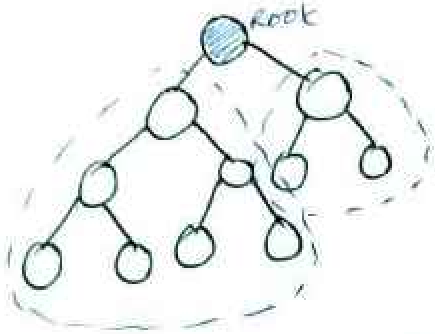
Apply BuildHeap on following array :-

A = 5, 3, 17, 10, 84, 19, 6, 22, 9



* Build heap starts at 4^{th} index.

* 

* 

$n = 11$

left subtrees = 7
Right subtrees = 3

→ The subtree of a node have at most ($^{2n}/_3$) node. The worst case occurs when the last row of the tree is exactly half full.

* Heapify RR $T(n) = T\left(\frac{2n}{3}\right) + O(1)$
(use MT to solve above RR)

$a = 1, b = 3/2$

$n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$

Apply 2nd case of MT

2nd case - $f_n \in \Theta(n^{\log_b a} \log^k n)$

then
$T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

$1 = \Theta(1 \log^k n)$ (valid for $k = 1$)

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

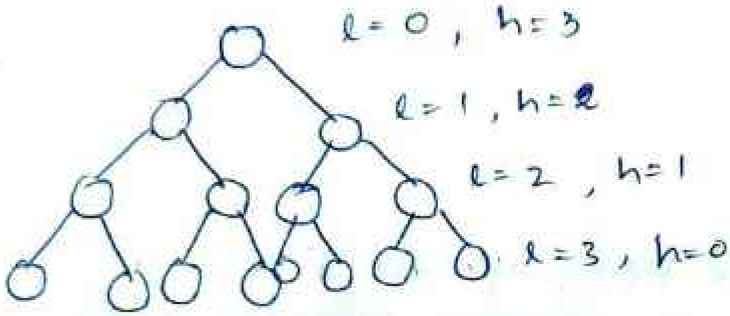
* Buildheap time complexity

$O(n \log n)$

* This is upper bound and is not asymptotically tight.

* Time for heapify varies with the height of the node in the tree, and the ~~bright~~ heights of the most nodes are small.

*



Property:- In an n-element heap, there are at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes of height h.

Eg:-
$n = 15$

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

nodes w height two $= \left\lceil \frac{15}{2^{2+1}} \right\rceil = 2$

* complexity of heapify - $O(1)$

* Complexity of Buildheap (another way)

$$T(n) = \sum_{h=0}^{\log n} \left[\frac{n}{2^{h+1}}\right] * O(h)$$

$$T(n) = O\left(n * \sum_{h=0}^{\log n} \frac{h}{2^{h+1}}\right) \quad \text{—} \ 1$$

$$T(n) \leq O\left(n * \sum_{h=0}^{\infty} \frac{h}{2^h}\right)$$

Some formulae:-

$$\sum_{n=0}^{\infty} x^n = \frac{1}{1-x} \quad \text{—} \ 2$$

Differentiating eqn 2 & multiplying of both sides by x

$$\sum_{n=0}^{\infty} n x^n = \frac{x}{(1-x)^2} \quad \text{—} \ 3$$

In eqn (1) put the value of from eqn 3

$$T(n) \leq O\left(n * \sum_{h=0}^{\infty} h * \left(\tfrac{1}{2}\right)^h\right)$$

$$T(n) \leq O\left(n * \frac{(1/2)}{(1-1/2)^2}\right) = O\left(n * \frac{1/2}{1/4}\right) = O(2n)$$

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

So, build heap can be implemented in $O(n)$.

(Amortized Analysis)

* HeapSort (A)

Build Heap(A) —— O(n)

for i = length(A) down to 2 —— O(n)

Swap (A[1], A[i])

heapify(A) = heapsize (A) − 1

heapify (A, 1) —— O(log n)

Time − O(n + n log n) = O(n log n)

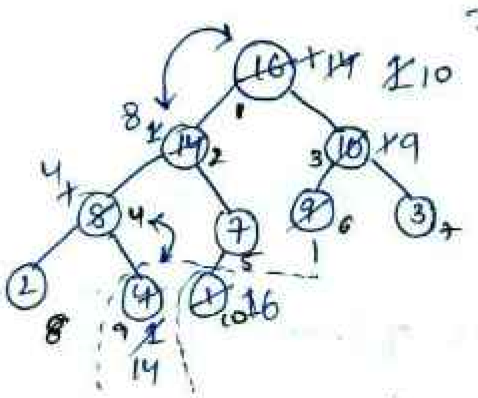
Extra space taken in recursion only so we can say that heap sort is inplace sorting algo.

Ex:−

16	14	10	8	7	9	3	2	4	1

Is it a heap? Yes

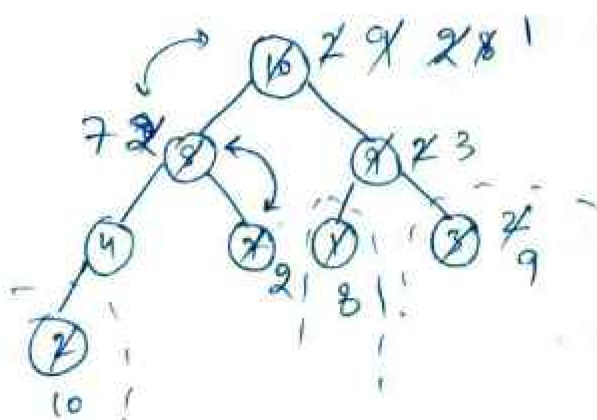
Step 1



10	8	9	4	7	1	3	2	14	16

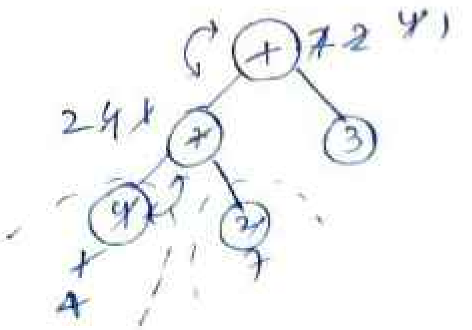
→ Eliminated (sorted.

Step 2



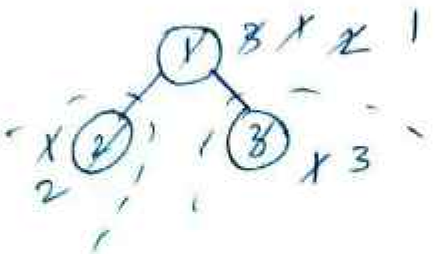
1	7	3	4	2	8	9	10	14	16

Step 3



1	2	3	4	7	8	9	10	14	16

Step 4



1	2	3	4	7	8	9	10	14	16

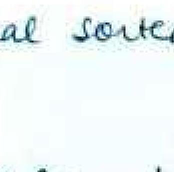
final sorted array after Heap Sort.

* Heap can be used as priority Queue.

→ Priority Queue :- PQ is a datastructure for maintaining a set S of elements each with an associated value called a key.

Operation on PQ :-

① Insert (S, x) :- insert a key.



— O(log n)

② Maximum (S) :- O(1) → return element.

③ Extract-max(S) :- Remove & return element.

O(log n)

Application :-

① Priority scheduling OS

② Graph algo
(Dijkstra, Prim's Algo)