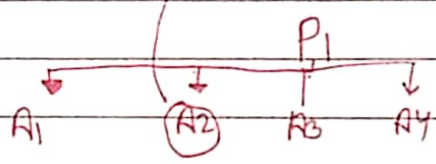
S. No.	Date	Title	Page No.	Teacher's Sign / Remarks

Algorithm

Problem
⇓
Program in C

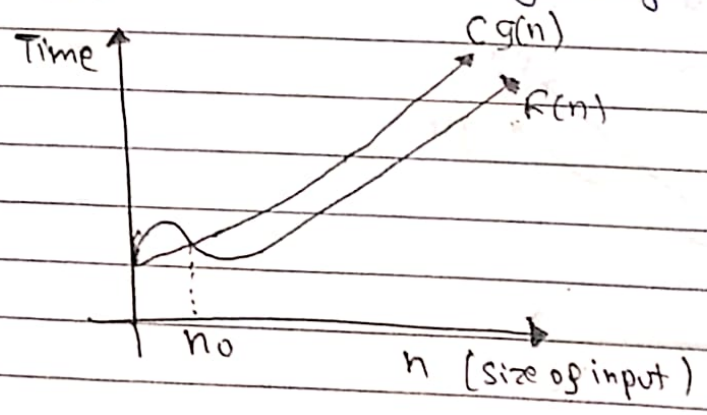


Design & analysis of algorithm:

One problem has many sol^n, we have to choose that sol^n which takes less memory & less time.

Asymptotic notation: It is mathematical tool to represent time complexity of algorithm. It is used the efficiency of Program that doesn't depend on machine.

① Big(Oh): We will try to find upper bound of f(n) ~~least~~



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

$n \geq n_0$

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

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

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

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

$3n+2 \leq cn$

$c = 4$

$3n+2 \leq 4n$

$n \geq 2$

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

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

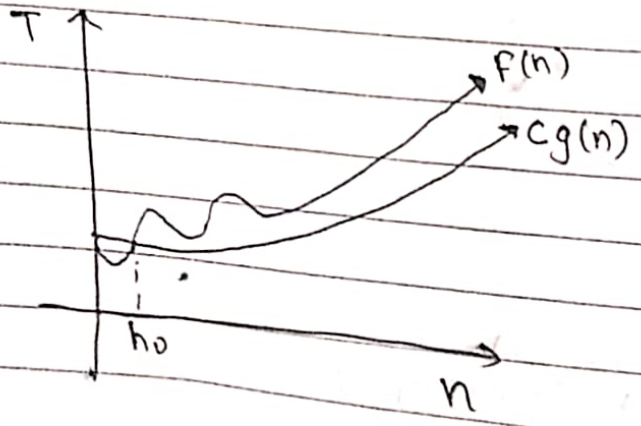
$g(n) = n^2$

$= n^3$

$= 2^n$

$= n^n$

Big Omega Ω : We have to make a lower bound of $f(n)$. Tightest lower bound we have to find.



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

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

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

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

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

$3n+2 \geq cn$

$3n+2 = \Omega(n)$ $\quad c=1,\ n_0 \geq 1$

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

$3n+2 \stackrel{?}{=} \Omega(n^2)$

$3n+2 \geq cn^2$, n_0 not possible for any c value.

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

$\log(n)$

$\log(\log(n))$

But we got for the lower bound which is most closer to it. That's why n.

* Big theta Θ: We have to bound a f^n in both upper & lower bound

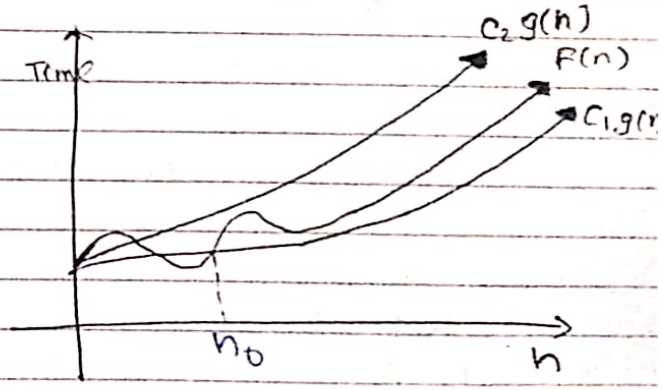
$f(n) = \Theta(g(n))$

$C_1 g(n) \leq f(n) \leq C_2 g(n)$

$C_1, C_2 > 0$

$n \geq n_0$

$n_0 \geq 1$



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

$f(n) \leq C_2 g(n)$

$3n+2 \leq 4(n)$, $n_0 \geq 1$

$f(n) \geq C_1 g(n)$

$3n+2 \geq n$, $n_0 \geq 1$

* $O(n)$: Worst Case, value can't be greater than that

* $\Omega(n)$: Best Case; Least time possible

* $\Theta(n)$: Average Case

When Best = Worst Case, on that Case we go for $\Theta(n)$

* | 5 | 7 | 1 | 2 | 4 | 8 | 11 |

$\Omega(1)$
$O(n)$
$\Theta\left(\frac{n}{2}\right) = \Theta(n)$

Algorithm

Iterative

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

Recursive.

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

* They both are equal in power, i.e Iterative can be converted into recursive & vice versa

* But analysis wise both are different.
Iterative is analysis by n & recursive by n/2 in the above model.

* If any program which doesn't have either recursive or iterative, then i.e its value doesn't depend upon input size so, its O(1).

```
A()
{
int i, j
for (i=1 to n)        O(n²), n² time ravi will print.
for j=1 to n
Pf (ravi)
}
```

* A()
```
{ i=1, S=1;
while (S<=n)
{ i++;
 S=S+i;
 Pf("ravi");
}}
```

S	1	3	6	10	15	. . .	n
i	1	2	3	4	5	. . .	K

(100 P)

They are sum of first 'n' natural

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

$$K^2 > n$$

$$K > O(\sqrt{n})$$

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

	1	4	9	...	n
i =	1	2	3	....	K

$K^2 > n$

$K > O(\sqrt{n})$

```
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 time	j = n time
K = 100 time	K = 2*100	K = 3*100	K = n*100

→ $100 + 2\times100 + 3\times100 - - - n\times100$

$= 100\,(1+2+3+ - - n)$

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

$= O(n^2)$

* 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	j=9	$j=n^2$
$k=\frac{n}{2}$	$k=\frac{n}{2}\times4$	$k=\frac{n}{2}\times9$	$k=\frac{n}{2}\times n^2$

$\frac{n}{2} + \frac{n}{2}\times4 + \frac{n}{2}\times9 + - - \frac{n}{2}\times n^2$

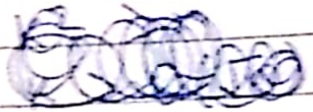
$\frac{n}{2}\left[1 + 2^2 + 3^2 + - - - n^2\right]$

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

$O(n^4)$

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

5	1	4	8	k	n
loop	1	2	5	4	..k



$2^k = n$

$k = \log_2 n$

$O(\log_2 n)$

```
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");
}
```

$= \frac{n}{2} \times \frac{n}{2} \times \log_2 n$

$O(n^2 \log_2 n)$

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

$\frac{n}{2} \times \log_2 n \times \log_2 n$

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

* Assume $n \geq 2$

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

$n = 2^k$

$k = \log_2 n$

$O(\log_2 n)$

* A()
{
for (i = 1 ; i<=n ; i++)
  for (j = 1 ; j<=n ; j = j + i)
    PF ("Hello");
}

i = 1	i = 2	i = 3	. .	i = n
j = 1 to n	j = 1 to n	j = 1 to n		j = 1 to n
n times	n/2 time	n/3		n/n

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

$= n(\log n)$

$= O(n \log n)$

```
A()
{ int n = 2^k ;
For (i=1 ; i<=n ; i++)        - n
{
  j = 2
  while (j<=n)
  { j = j^2 ;
    PF (aul.) ;
  }
}
```

k = 1	k = 2	k = 3
n = 4	n = 16	$n = 2^8$
j = 2, 4	j = 2, 4, 16	$j = 2, 2^2, 2^4, 2^8$
n x 2 time	n x 3 time	n x 4 time.

$n \times (k+1)$

$n = 2^{2^k}$

$k = \log \log n$

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

$O(n \log \log n)$

Recursive:

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

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

constant time.

Back Substitution:

```
A(n)
{
  if (n>1)
    return (A(n-1))
}
```

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

$T(n) = 1$; $n = 1$

$T(n) = 1 + T(n-1)$ — ①

$T(n-1) = 1 + T(n-2)$ — ②

Put ② in ①

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

$= 2 + T(n-2)$

$= 3 + T(n-3)$

$= k + T(n-k)$

$(n-k = 1, \quad k = n-1)$

$= (n-1) + 1$

$= n$

So $O(n)$

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

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

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

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

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

$n - k = 1$

$k = n-1$

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

$\Rightarrow n + (n-1) + (n-2) + \cdots 2 + 1$

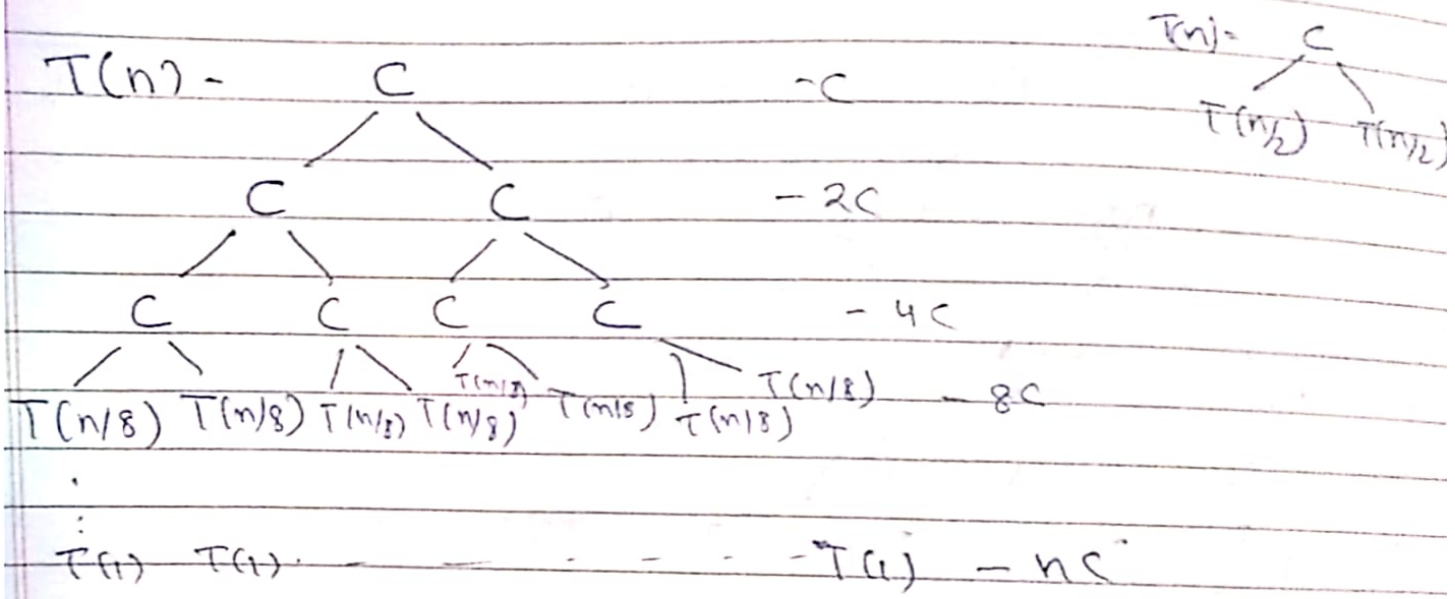
$\Rightarrow$ Sum of I^{st} n term

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

$= O(n^2)$

Recursion tree METHOD:

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



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

$(n = 2^k)$

$c + 2c + 4c + \cdots nc$

$= c(1 + 2 + 4 + \cdots 2^k)$

$$c\left(\frac{1 + (2^{k+1} - 1)}{2 - 1}\right)$$

$= c(2^{k+1} - 1)$

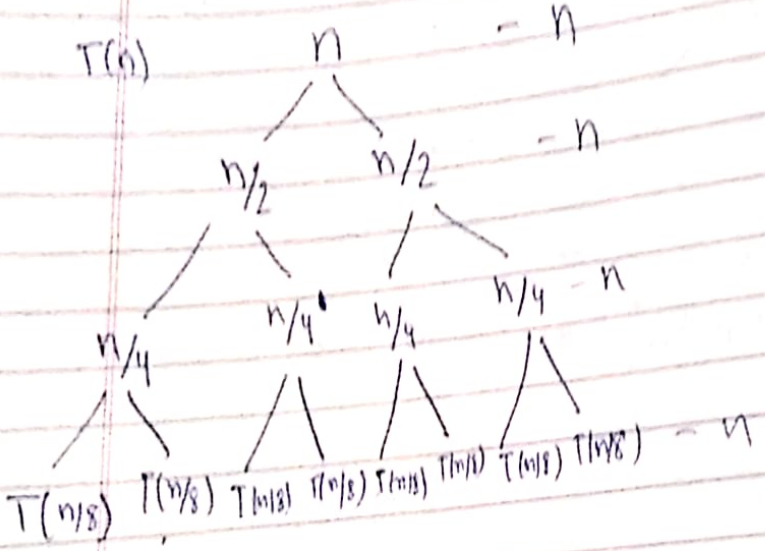
$= c(2n - 1)$

$O(n)$

then relation is in $\sqrt{n}$ then it will take $O(\log n)$

* Recursion tree method

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



$\Rightarrow$ $n + n + n + \cdots n$

Total number $\log_2 n + 1$

$= O(n \log_2 n)$

Master's theorem:

- Comparing two f^n:

n^2 n^3

cancel out n^2

$1 < n$

Compare 2^n n^2

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

n $2 \log_2 n$

Put $n = 2^{100}$

$2^{100} > 200$

$2^n > n^2$

Compare 3^n 2^n

$n \log_2 3$ $n \log_2 2$

$\log_2 3 > \log_2 2$

* n^2 $n \log n$

n $\log n$

$n = 2^{100}$

$2^{100} > 100$

$n^2 > n \log n$

n $(\log n)^{100}$

$\log n$ $100 \log \log n$

Put $n = 2^{128}$

128 100×7

700

$n = 2^{1024}$

1024 100×10

$1024 > 1000$

Hence

$n > (\log n)^{100}$

* $n^{\log n}$ $n \log n$

$\log n \log n$ $\log n + \log \log n$

$n = 2^{1024}$

1024×1024 $1024 + 10$

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

* $\sqrt{\log n}$ $\log \log n$

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

$n = 2^{1024}$

$\frac{1}{2} \times 10$ $\log 10$

$5 > 3.2$

* $n^{\sqrt{n}}$ $n^{\log n}$

$\sqrt{n} \log n$ $\log n \cdot \log n$

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

$n = 2^{1024}$

$2^{512} > 1024$

* $f(n) = \begin{cases} n^3 & 0 < n < 10,000 \\ n^2 & n \geq 10000 \end{cases}$

$g(n) = \begin{cases} n & 0 < n < 1000 \\ n^3 & n > 100 \end{cases}$

We will alway choose the fn which gives us larger value at infinite

	0-99	100-9999	10000
F(n)	n^3	n^3	n^2
g(n)	n	n^3	n^3

$n^3 > n^2$, $n > 10000$

$g(n) > f(n)$

$f(n) = O(g(n))$, $n_0 = 10,000$

$f_1 = 2^n$, $F_2 = n^{3/2}$, $f_3 = n \log n$, $f_4 = n^{\log n}$

2^n $n^{3/2}$

$n \log 2$ $\frac{3}{2} \log n$

$n > \frac{3}{2} \log n$

$n^{3/2}$ $n \log n$

$\frac{3}{2} \log n$ $\log(n \log n)$

$n = 2^{1024}$

$\frac{3}{2} \times 1024$ $\log(2^{1034})$

$\frac{3}{2} \times 1024 > 1034$

$n^{3/2}$ $n^{\log n}$

$\frac{3}{2} \log n$ $\log n \times \log n$

$n = 1024$

$\frac{3}{2} \times 1024 < 1024 \times 1024$

2^n $\quad$ $n \log n$

$n \log 2$ $\quad$ $\log n \times \log n$

2^{1024} $\quad > \quad 1024 \times 1024$ $\quad n = 1024$

Hence $2^n > n^{\log n} > n^{3/2} > n \log n$

Master's Algorithm:

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

$a \geq 1$, $b > 1$, $k \geq 0$ & p is real number.

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. $p < 0$, then $T(n) = \theta(n^k)$

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

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

$a \quad b^k$

$3 < 2^2$

Here $p = 0$

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

$= \theta(n^2)$

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

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

$a \quad b^k$

$4 = 4$

$p = 0$

2-a

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

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

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

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

$a < b^k$

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

$= \theta(n^2)$

4) $T(n) = 2^n T(n/2) + n^n$ x Not applicable,

$2^n = a$, It should be constant.

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

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

$a > b^k$

$16 > 4^1$

$T(n) = \theta(n^{\log_4 16})$

$= \theta(n^2)$

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

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

$a = b^k$

$2 = 2^1$

$p = 1$

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

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

7)

2-b

8)

C

3-a-i

*

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

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

$a = b^k$

$2 = 2^1, \quad p = -1$

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

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

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

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

$a < b^{0.51}$

$p = 0$

$T(n) = \theta(n^{0.51} \log^0 n)$

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

$T(n) = 0.5\,T(n/2) + 1/n$

$a = 0.5, \quad b = 2, \quad K = -1, \quad p = 0$

Not possible by Masters theorem, because $a < 1$.

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

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

$a < b^k$

$6 < 3^2$

$p = 1$

$T(n) = \theta(n^2 \log^1 n)$

13

14) $T(n) = 64T\left(\frac{n}{8}\right) - n^2 \log n$

$a = 64, \quad b = 8, \quad k = 2, \quad p = 1$

* Not valid (-) is there according to master's

12) $T(n) = 7T\left(\frac{n}{3}\right) + n^2$

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

$a < b^k$

$7 < 3^2$

$p = 0$

15)

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

$= \theta(n^2)$

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

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

$a > b^k$

$4 > 2^0$

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

$= \theta(n^2)$

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

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

$a > b^k$

$\sqrt{2} > 2^0$

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

$\theta(\sqrt{n})$

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

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

$a > b^k$

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

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

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

$a > b^k$

$3 > 2^1$

$\Theta(n^{\log_2 3})$

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

$a = 3, \quad b = 3, \quad k = 1/2, \quad p = 0$

$a > b^k$

$3 > \sqrt{3}$

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

* $T(n) = 4T(n/2) + cn$

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

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

$T(n) = 3T(n/4) + n \log n$

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

$a < b^k$

$3 < 4$

$\Rightarrow \Theta(n^1 \log^1 n)$

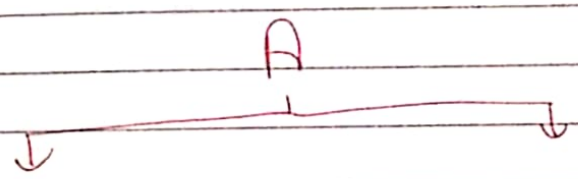
$= \Theta(n \log n)$

Space Analysis:

$(n) = O(n)$ Size of input space required.

$= O(1)$ Constant space required

$= O(n^2)$



Iterative Recursive.

Space complexity: The extra space which have created to solve an algorithm. we have to decrease that space.

Given array → Input size

```
Algo( A, i, n)
{
 int i, j=0;          ⟵ They are the extra space which we have created
 for(i to j)          Total = 2
  A[i] = 0;           O(1)
}
```

* Algo(A, l, n)

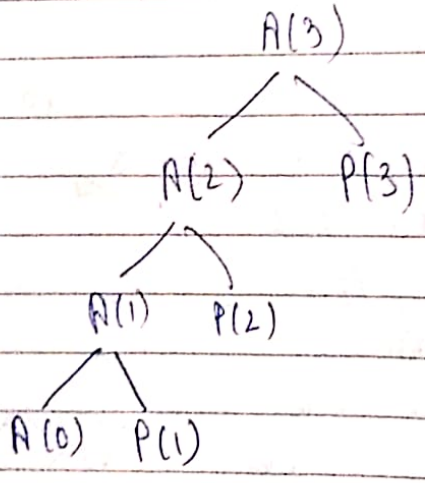
```
{
int i;
Create B[n];          i + B[n]
for (i=1 to n)        1 + n
B[i] = A[i];          O(n)
}
```

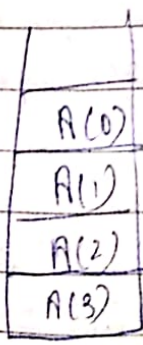
* Algo(A, l, n)

```
{
Create B[n,n]. - n^2
int i,j ;      - 2
for (i=1 to n)                n^2+2
  for (j=1 to n)              O(n^2)
  B[i,j] = A[i];
}
```

* A(n) ~~O(n+1)~~

```
{
if (n>=1)
{ A(n-1)
  PF(n)
}
}
```





Space complexity: Height of stack or no of recursive call.

$(n+1) * K$

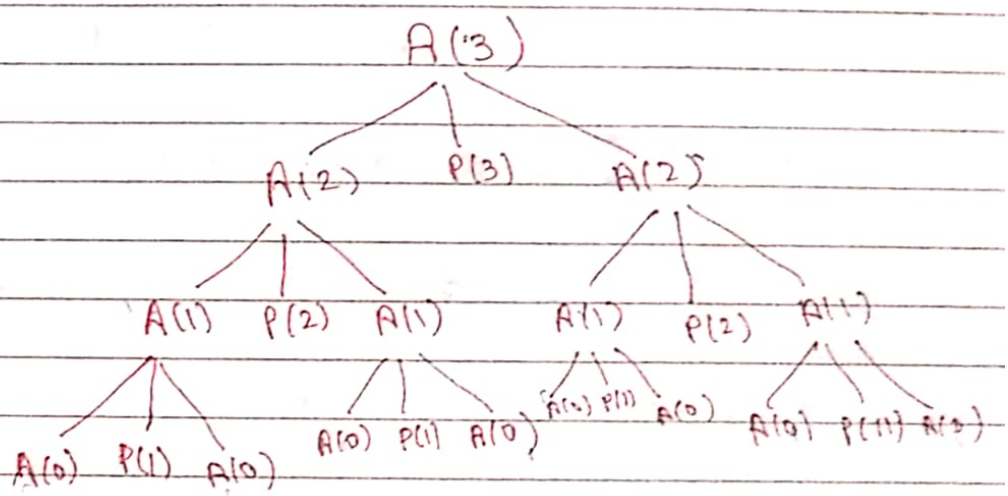
$O(n)$ = Space complexity

Time Complexity:

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

$$T(n) = 1 + 1 + 1 + \cdots \quad T(0)$$
$$= 1 + 1 + 1 + \cdots - 1$$
$$= n + 1$$
$$= O(n)$$

```
A(n)
{
 if(n >= 1)
 {
   A(n-1);   1
   Pf(n)     2
   A(n-1);   3
 }
}
```



Output 1 2 1 3 1 2 1

Total no of recursive call = 15 (no of time a call)

$A(3) = 15 \rightarrow 2^{3+1} - 1$
$A(2) = 7 \rightarrow 2^{2+1} - 1$
$A(1) = 3 \rightarrow 2^{1+1} - 1$

Space complexity - Depth of tree

$(n+1) * K \rightarrow$ size of 1 stack

$= O(n)$

Time complexity: $O(2^{n+1}) = O(2^n)$

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

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

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

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

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

$\Rightarrow 2^3 T(n-3) + 2^2 + 2 + 1$

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

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

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

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

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

Amortized Analysis:

In an amortized analysis we average the time required to perform a sequence of data structure operation over all the operation performed.

Using an amortized analysis, we can show that the average cost of an operation is small, eventhough a single operation within the sequence might be expensive

Used to analyze time complexities of hash tables, splay trees & Disjoint sets.

Technique used in amortized analysis
- Aggregate Method
- Accounting Method
- Potential Method

Aggregate Analysis:

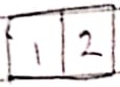
Hash table insertion:

↳ Increase the size of table whenever it become full.
Allocate memory for a large table of size double the old table.
Copy the contents of old table into new table.
Free the old table.

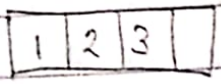
Initially table is empty & size is 0.

Insert 1 (overflow)

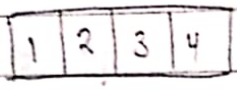
Insert 2 (overflow)



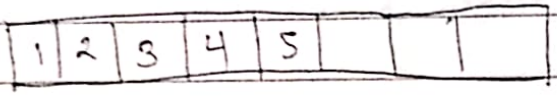
Insert 3 (overflow)



Insert 4



Insert 5 (overflow)



Item no.	1	2	3	4	5	6	7	8	9	10
Table size	1	2	4	4	8	8	8	8	16	
cost	1	2	3	1	5	1	1	1	9	

(1 move 1 insert) (2 move 1 insert)

In normal case it take = n (element to insert) $\times$ $O(n)$ (for all element)

$= O(n^2)$

But according to aggregate analysis.

The amortized cost $= \frac{(1+2+3+1+5+1+1+1+9+\cdots)}{n}$

$= \frac{(1+1+1+1+1\cdots+1) + (1+2+4+8+\cdots \lfloor \log_2 n-1 \rfloor + 1 \text{ times})}{n}$

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

The amortized cost $= O(1)$

Inplace

Sorting Algorithm:

```
Insertion-Sort(A)                    // Iterative model
{
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
}
```

	0	1	2	3	4	5	6	7	8	(Index) in array
		9	6	5	0	8	2	7	1	3
		6	9	5	0	8	2	7	1	3
		5	6	9	0	8	2	7	1	3
		0	5	6	9	8	2	7	1	3
		0	5	6	8	9	2	7	1	3
		0	2	5	6	8	9	7	1	3
		0	2	5	6	7	8	9	1	3
		0	2	5	6	7	8	9	3	
		0	2	3	5	6	7	8	9	

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

↙ comparison Movement

1 → 1 + 1 = 2

2 - 2 + 2 = 4

3 - 3 + 3 - 6

:

n → n + n - 2n

$2 + 4 + 6 + \cdots 2n$

$2[1 + 2 + 3 + \cdots n]$

$n(n+1) = O(n^2)$ = Time complexity.

Best case $\Omega(n)$

1 2 3 4 5 6 7 8 9

we have to only compare them, no movement, because they are already sorted.

$1 + 1 + 1 + \cdots 1$

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

Space complexity = $O(1)$

key, i, j take the 3 space, ie constant space.

Now trying to reduce time complexity:

Before that element, everyone is sorted.

	Comparison	movement	
Binary	$O(\log n)$	n	$O(n)$
Doubly linked list	$O(n)$	$O(1)$	$O(n)$

Not able to reduce Time complexity.

It is still $O(n^2)$, space complexity $O(1)$

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

n ↓ No of comparison, $O(n)$ ↓ no of traverse.

Out of place : Extra space required

MERGE SORT

MERGE (A, p, q, r) → Array

{

$n_1 = q - p + 1$

$n_2 = r - q$

let $L[1 \cdots n_1 + 1]$ AND $R[1 \text{ to } n_2 + 1]$ be new array

For (i = 1 to n_1)

$L[i] = A[p + i - 1]$

For (j = 1 to n_2)

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

$L[n_1 + 1] = \infty$

$R[n_2 + 1] = \infty$

i = 1 , j = 1

For (k = p to r)

if ($L[i] \leq R[j]$)

$A[k] = L[i]$

i = i + 1

else $A[k] = R[j]$

j = j + 1

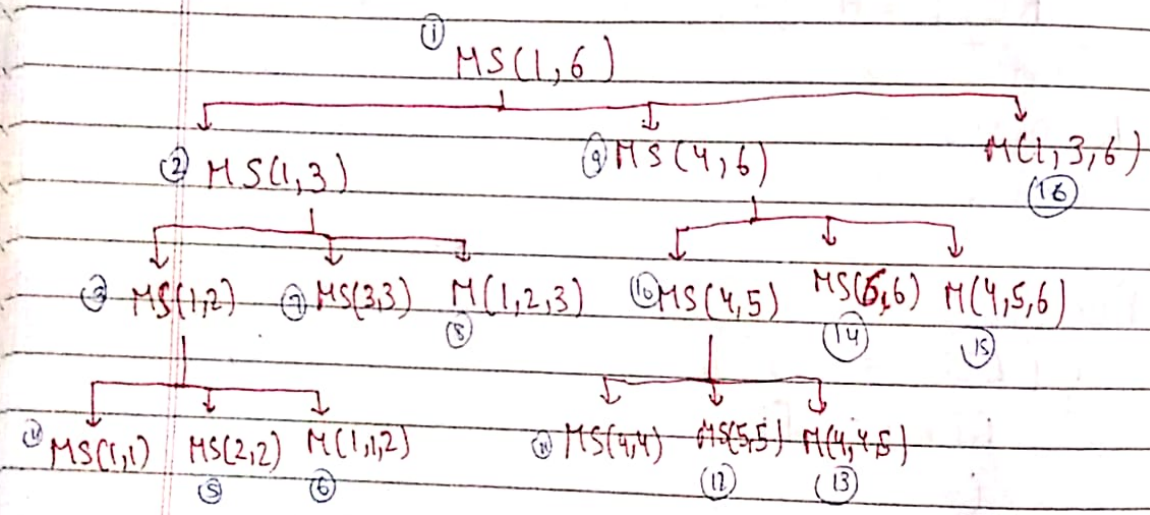
}

Space complexity O(n), because we need one more array of that size to keep it in them.

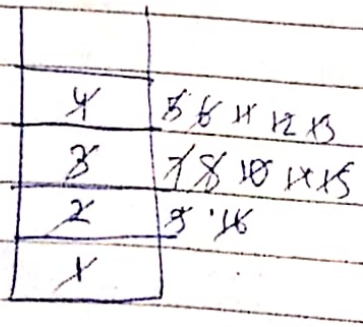
Now real algo:

```
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)
}
```

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



There are total 16 fⁿ call.
So space required for this call is



we required only 4 stack for the p^{th} call.

$(\lceil \log_2 n \rceil + 1)$ level (stack required)

$n = 6$

n-space is required for Merge call.

Merge - O(n)
Stack - O(log n)

→ O(n + log n)
[- O(n)] = Space required in merge sort.

Now, 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) (Time required to merge them, because we merge the all element)
}
```

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

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

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

$a = b^k$, $\Theta(n^{\log_b^a} \log^{p+1} n) = \Theta(n \log n)$

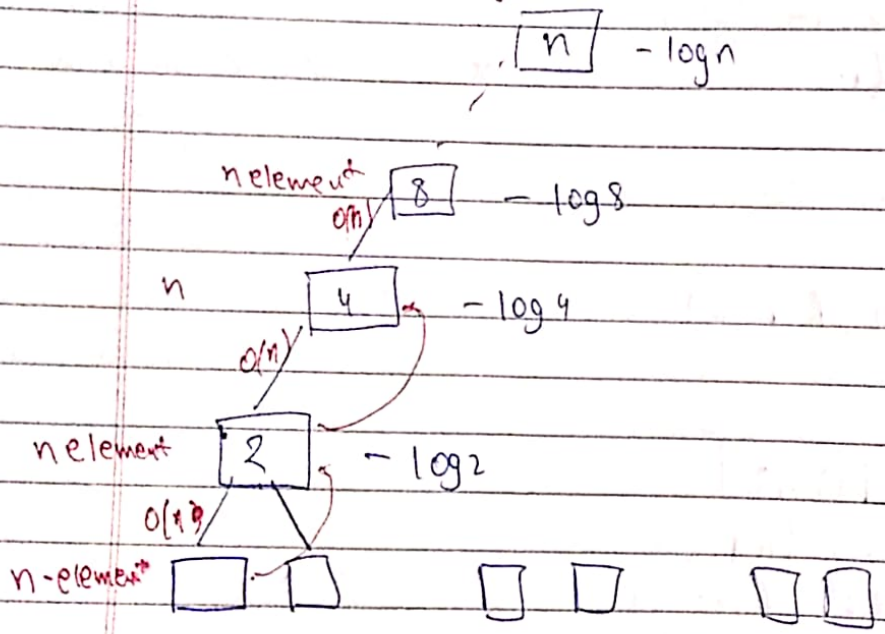
* In other way we can find it.

* At every level of tree we have to do total n-comparison

* The total level in a tree is log n

* So total time require will be (n log n)

* Given n elements, merge them into one sorted list using merge procedure.



Total level = log n

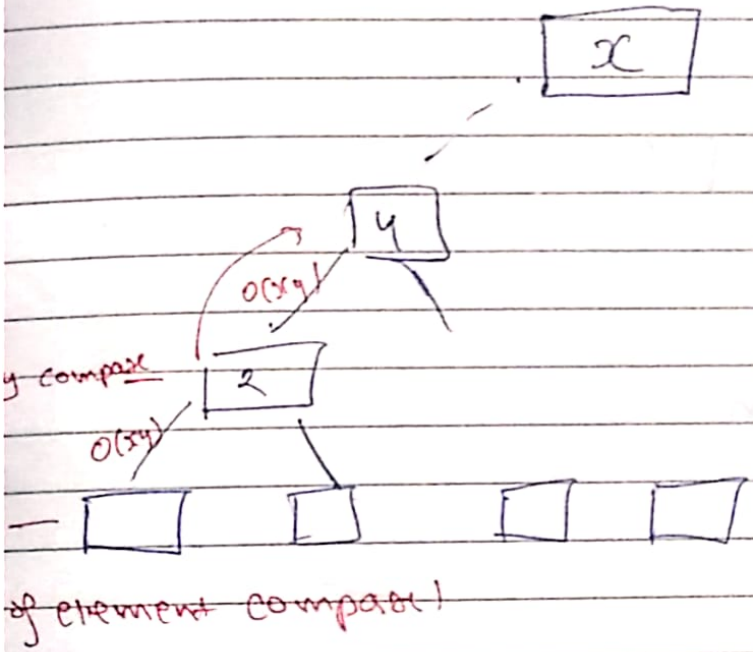
So total time complexity = O(n log n)

Given "logn" Sorted list each of size "n/logn". What is the total time required to merge them into one single list

Assume - $x = \log n$ (no of list)

$y = \frac{n}{\log n}$ (Size of list or total no of element in one list)

So total no of element = $x \times y$

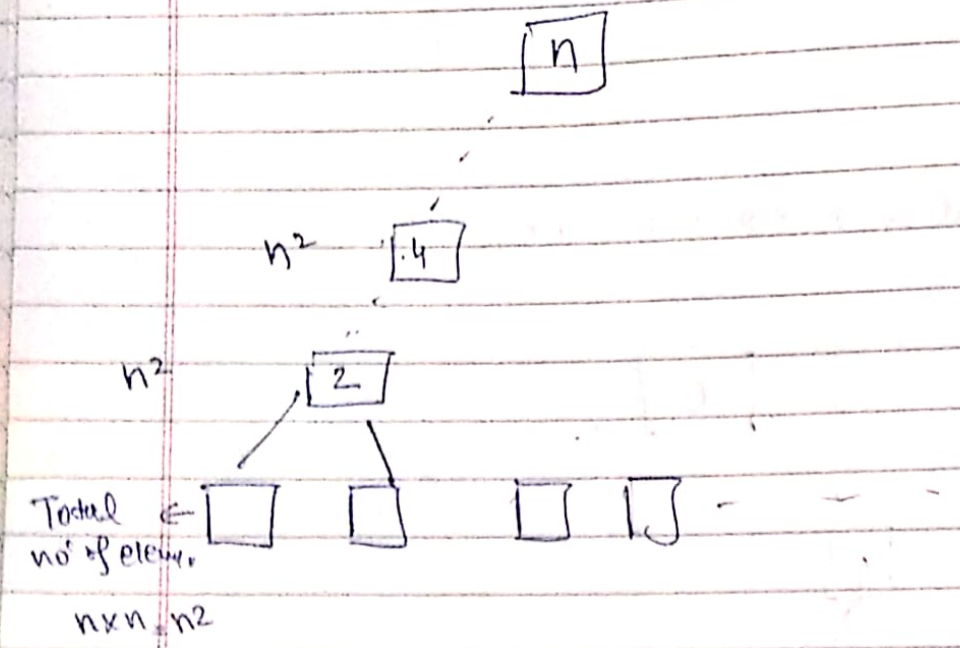


$xy \times \log x$

$\log n \times \frac{n}{\log n} \times \log \log n$

$O(n \log \log n)$

* n string each of length 'n' are given. Then what is the time taken to sort them. the 2 way merge



Total level = $\log n$

* $O(n^2 \log n)$ → level of tree.
↓ Total no of element

* Merge sort uses divide & conquer.

Result after 2 pass of merging:

20 47 15 8 9 4 40 30 12 17

(20 47) (8 15) (4 9) (30 40) (12 17)

8 15 20 47 4 9 30 40 12 17

QUICK SORT:

```
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
}
```

p & r are the boundary of an array.

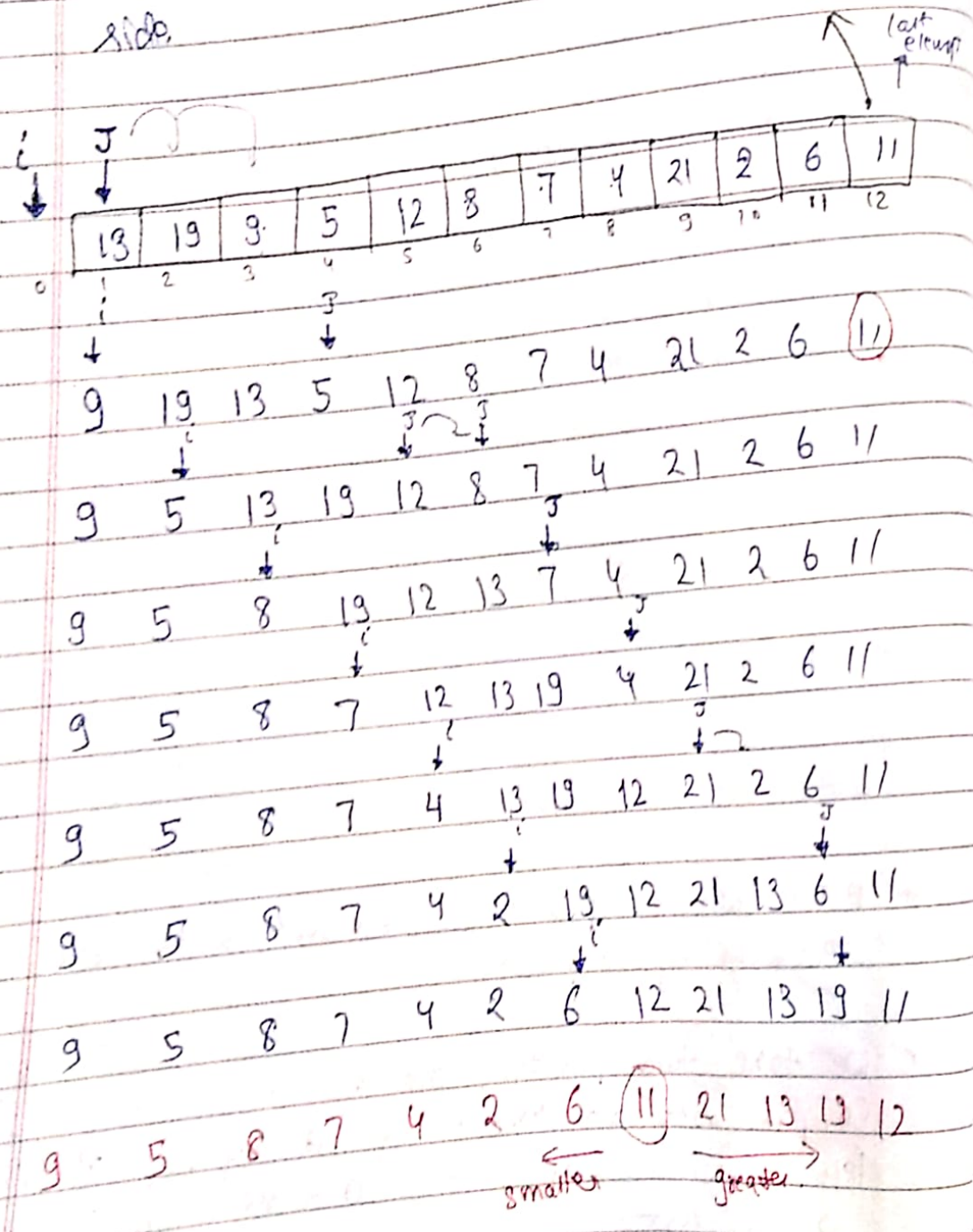
* We will compare every element with the last element of an array.

We take two pointer, one will increment & compare with last element of array, if it is less it will swap with another pointer of an array which is behind him.

In this way we find the appropriate

position of last element, & we put it into its place.

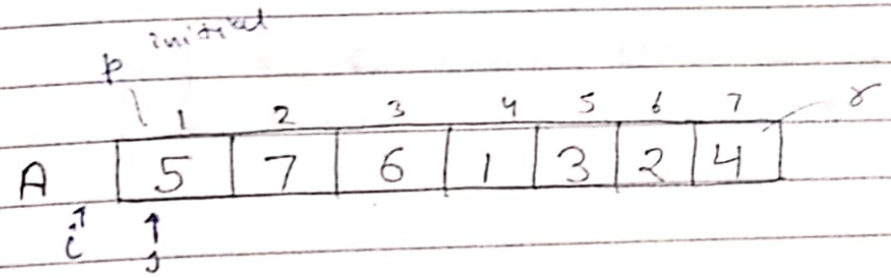
* The element which is left to it is smaller than him & greater then it will be in right side.



* Any element left to i, is smaller then last element.
* Every element b/w i & j are greater then

last element.

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



1 7 6 5 3 2 4

↑i ↑j

1 3 2 5 7 6 4

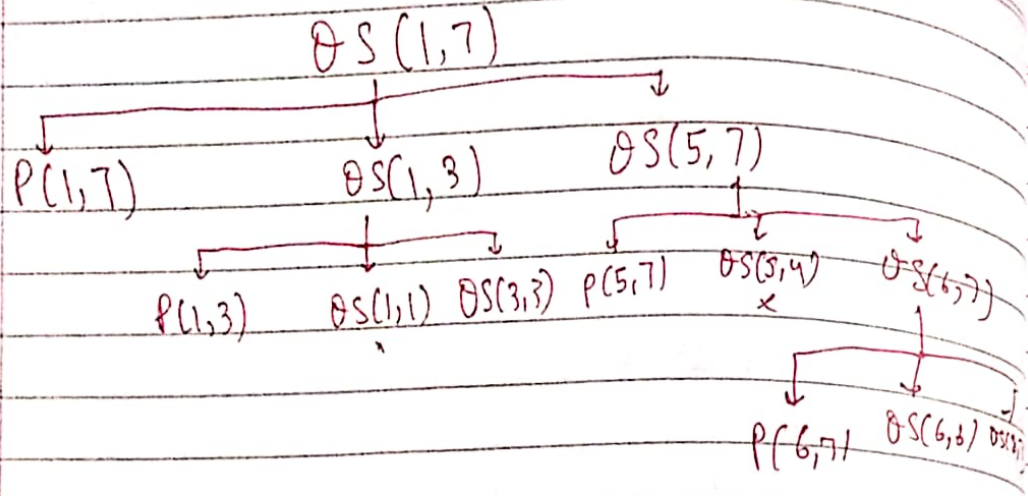
↑i ↑j

(1 3 2) 4 (7 6 5)

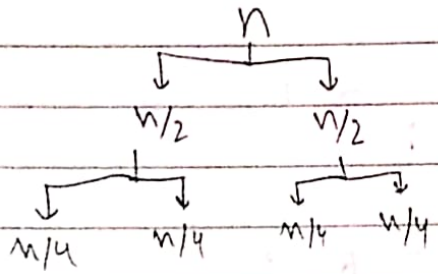
i j r

(1) 2 (3) 4 (5 (6) 7)

No Extra space required.

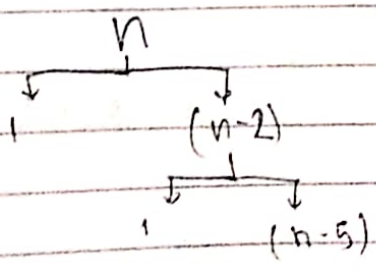


We don't need extra space for partition. But for free we need space in form of stack.



* $O(\log n)$ - Space complexity.

* But in worst case it can be $O(n)$



	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)	$T(n/2)$	$T(n-1)$
}		
}		

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

$= \theta(n \log n)$

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

$= T(n-1) + cn$

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

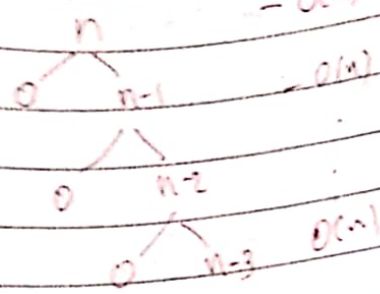
$= c + c2 + c3 + \cdots cn$

$= O(n^2)$

* If the given sequence is either in ascending or descending order, then time complexity will be

$O(n^2)$

Even the all no's are same, time complexity will be $O(n^2)$

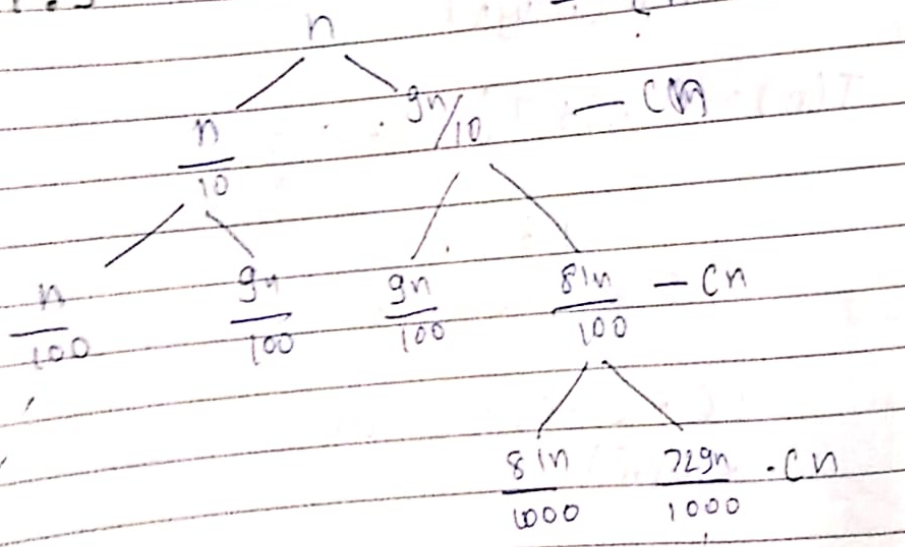


$O(n)$
$O(n)$
$O(n)$

Total level = $O(n)$

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

* Let us suppose that splitting are in the ratio of 1:9



cn
cn
cn
cn
$< cn$
$1 < cn$

$$\frac{n}{1} - \frac{n}{10/9} - \frac{n}{(10/9)^2} \quad \cdots 1$$

$$\log_{10/9} n \cong \Theta(\log_2 n)$$

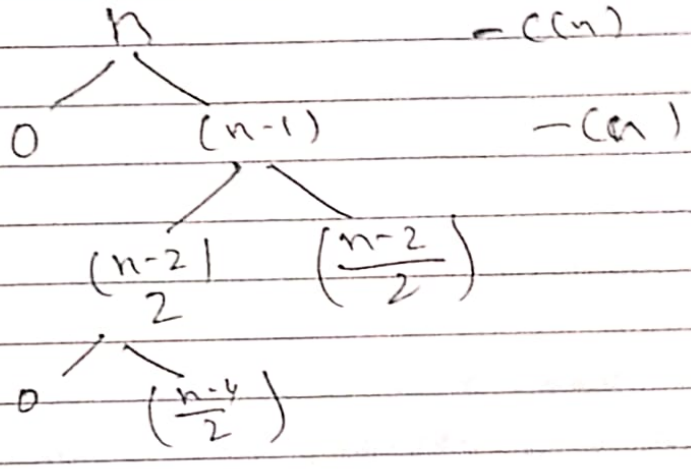
$\therefore$ Total level

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

If they are in any ratio complexity will be $\theta(n \log n$

* $(0, n-1)$ is the only splitting case where we get $\theta(n^2)$

* If the split is in the form of bad split followed by good split, then what will be the time complexity.



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

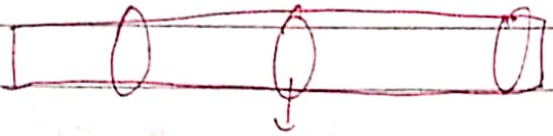
$$= O(n) + 2T\left(\frac{n-2}{2}\right)$$

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

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

Hence even in this case we are getting time complexity $\theta(n \log n)$

* The median of n element can be found in O(n) time. Which one of the following is correct about complexity of quick sort, in which median is selected as pivot?



In O(n) Find Median
O(1) Put it into last
O(n) for partitioning.
n/2 , n/2

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

We add those time, because at any level, first we find median, then independently we are doing comparison.

* The quick sort, for sorting 'n' elements the $(n/4)^{th}$ smallest element is selected as pivot using O(n) time algorithm. What is the worst case time complexity of quick sort?

$$T(n) = O(n) + O(1) + O(n) + T(n/4) + T(3n/4)$$
$$T(n) = O(n) + T(n/4) + T(3n/4)$$
$$= \Theta(n \log n)$$

1:3, they are dividing

Introduction to Heap:

	insert	Search	Find min	Delete Min
Unsorted Array	O(1)	O(n)	O(n)	O(n)
Sorted Array	O(n)	O(log n)	O(1)	O(n)
Linked list	O(1)	O(n)	O(n)	O(n)

Search in sorted list

1 4 6 (10) 20 30

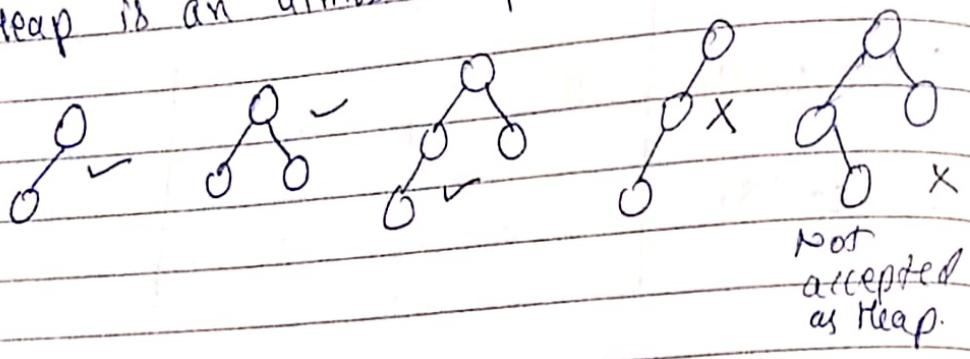
$T(n) = 1 + T(n/2)$

$= \Theta(\log n)$

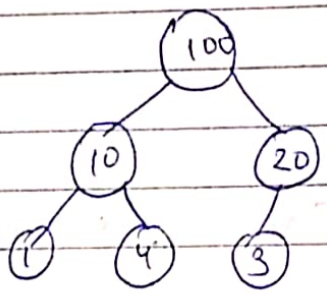
	insert	Search	Find min	Delete min
Min heap	O(log n)		O(1)	O(log n)

We mainly focus on insert, Find min & delete min in any algo. That's why Min heap is used.

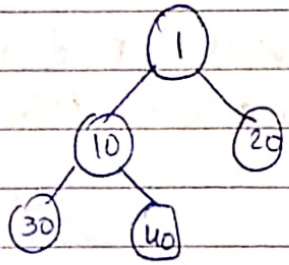
* Heap is a binary tree or 3ary tree - - - n-ary tree
* We fill the element from left to right
* Heap is an almost complete Binary tree



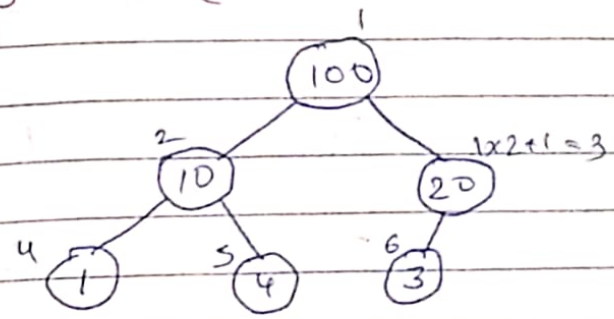
MAX Heap: It is a tree in which root node is maximum. In other way parent node should be greater than its child.



Min Heap: It is a tree in which root contain the min value.



MAX Heap element placement, multiply parent by 2, (x×2, x×2+1) are the position of children



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

With the help of child find parent index

$$\left\lfloor \frac{x}{2} \right\rfloor , \quad \left\lfloor \frac{5}{2} \right\rfloor = 2$$

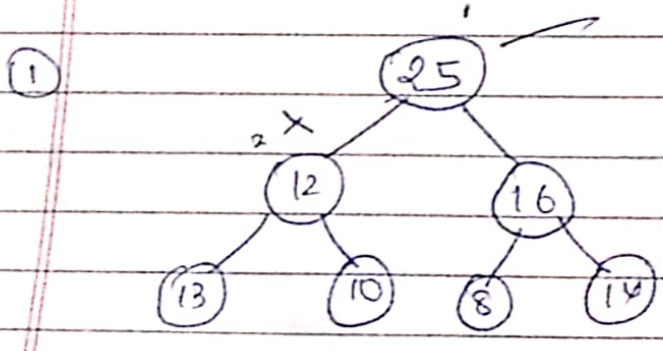
$LC(i) = 2 * i$
$RC(i) = 2 * i + 1$

$PC(i) = \lfloor i/2 \rfloor$

} These operation are easy to implement. There is only left shift.

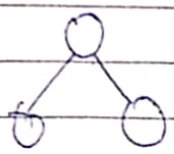
	A.length
25, 12, 16, 13, 10, 8, 14	7
25, 14, 16, 13, 10, 8, 12	7
25, 14, 13, 16, 10, 8, 12	7
25, 14, 12, 13, 10, 8, 16	7
14, 13, 12, 10, 8	5
14, 12, 13, 8, 10	5
14, 13, 8, 12, 10	5
14, 13, 12, 8, 10	5
89, 19, 40, 17, 12, 10, 2, 5, 7, 11, 6, 9, 70	13

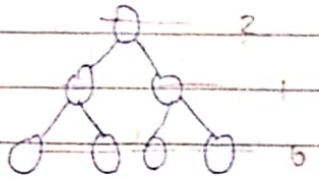
A.heapsize: How many node is satisfying max heap property. We start from first index go till that node, where our condition is dissatisfied.



A. Heapify

Complete binary tree





If height = h
then max no of node in a complete binary tree

H	1	2	3	4
MAX	3	7	15	31

$Max = (2^{H+1} - 1)$

Node is given, Find Height of tree

$H = \lfloor \log_2 N \rfloor$

If there is complete 3-ary tree,

H	1	2	3	4
MAX	1	4	13	40

Height = 2 = $3^2 + 3^1 + 3^0 \Rightarrow 13$

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

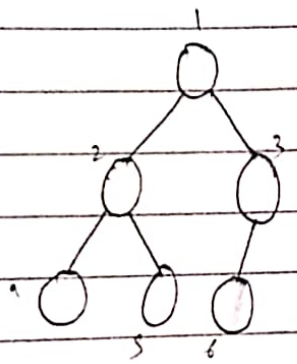
Node = $\lfloor \log_3 2N+1 \rfloor$

* In n-ary tree

$$MAX = \left(\frac{n^{H+1} - 1}{n-1} \right)$$

* If we sort the array, then we can build the Heap in $\theta(n \log n)$ time. Because we can apply merge sort here.

* But for building heap, we don't need to do sorting. $\theta(n)$ time we can build it.

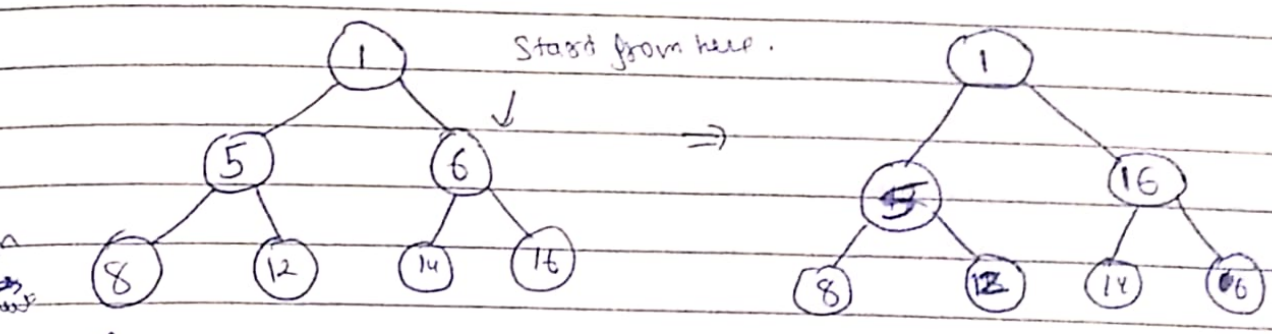


Leaf start to end $\left(\left\lfloor \frac{n}{2} \right\rfloor + 1 \text{ to } n \right)$

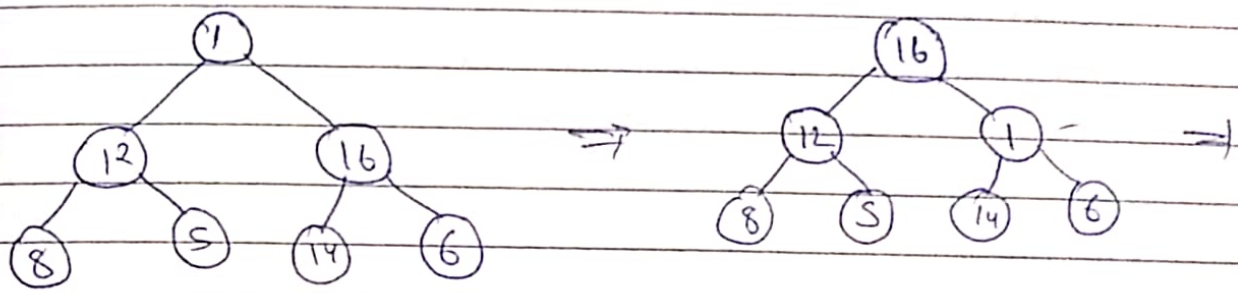
$\left\lfloor \frac{6}{2} \right\rfloor + 1$ to 6

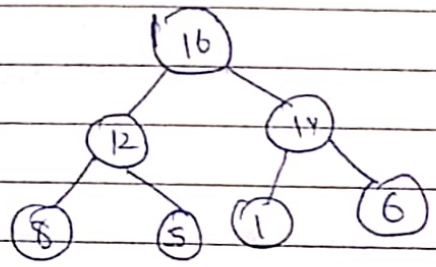
6 (4 to 6) are leaf

1 5 6 8 12 14 16



st from
from here
, if they are
already MAX, Heap





```
MAX-HEAPIFY(A,i)
{
l = 2i;
r = 2i+1;
if (l ≤ A.heapsize and A[l] > A[i])
    largest = l;
else largest = i;

if (r ≤ A.heapsize and A[r] > A[largest])
    largest = r;
if (largest ≠ i)
  exchange A[i] with A[largest]
  MAX-HEAPIFY(A, largest)
}
```

Time complexity in worst case $O(\log n)$

↓

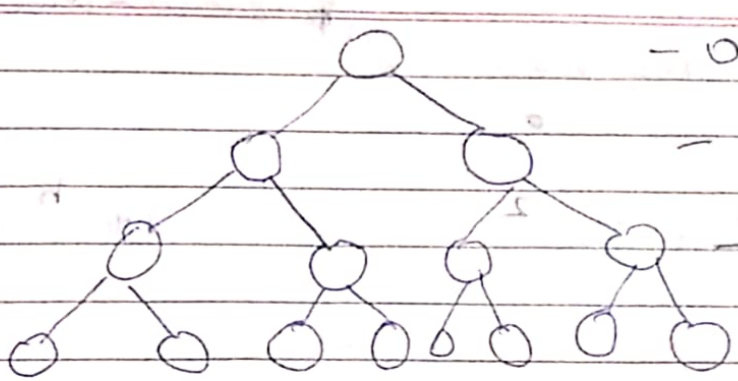
n = the node from ~~where we apply~~ heapify

Space complexity $O(\log n)$

no of recursive or p^n call.

```
BUILD-MAX-HEAP(A)
{
A.heapsize = A.length
For (i = ⌊A.length/2⌋ down to 1)
MAX-HEAPIFY(A,i)
}
```

(Started from Largest no leaf node)



- O(3)
- O(2) (Time required to balance)
- O(1)

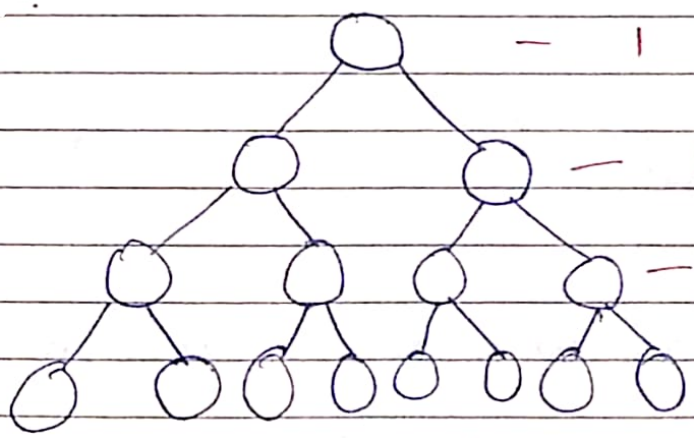
Finding node at any level = $\left\lceil \frac{n}{2^{H+1}} \right\rceil$

$$\sum_{h=0}^{\log n} \left\lceil \frac{n}{2^h \cdot 2} \right\rceil (ch)$$

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

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

$$< \frac{cn}{2} \times 2 = n = O(n)$$ Time Complexity.



- 1
- 2
- 3

→ 3 ⇒ Total no of time

we have to balance any node = Total height from that node (consider itself bottom) + 1 (except root node).

Total node at any level = $2^{(\text{Total height} - \text{level})}$

$= 1 \times 2^0 + 2 \times 2^1 + 3 \times 2^2 + \cdots (h+1) \times 2^h \rightarrow \theta$

- They all will be $\log n$ time present.
  Total no of element $\log n$

We can neglect 1, 2, 3 ... n+1, as they are very small as compare to squaring term.

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

$= 2^{\log n}$

$= n = O(n)$

* Heap-Extract-Max(A)

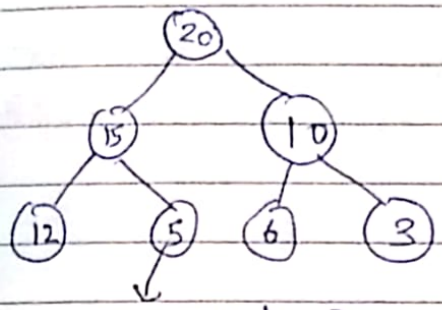
```
{
 if A.heapSize < 1
    error "heap underflow"
 max = A[1]
 A[1] = A[Heap_Size]
 A.heap_size = A.heap_size - 1
 MAX-HEAPIFY [A,1]
 return max
}
```

→ only extracting an key & then arranging them

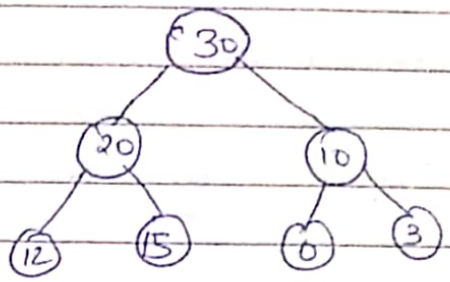
Time required to extra max : $O(\log n)$

Space required - $O(\log n)$

```
Heap-increase-key (A, i, key)
{
if (key < A[i])
      error
A[i] = key
while (i > 1 and [i/2] < A[i])
      exchange A[i] and A[i/2]
      i = i/2
}
```

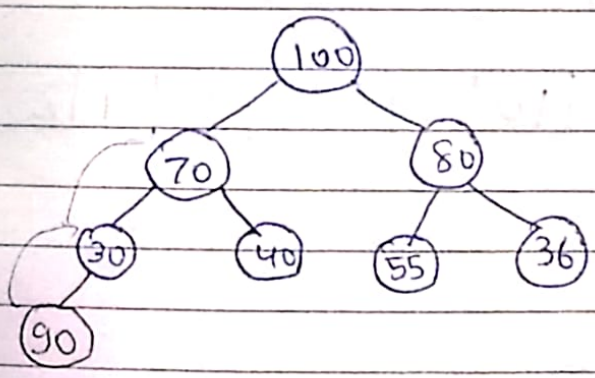


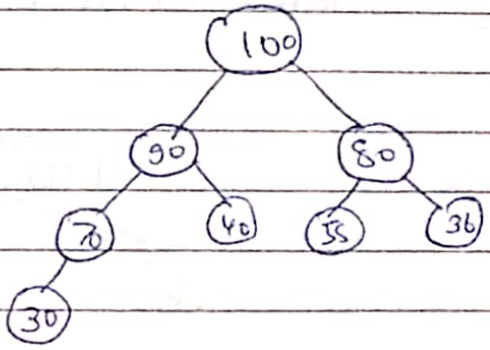
Increase to 30, then we have to balance it



Time complexity : O(log n)
Space complexity : O(1)

Insert element in heap





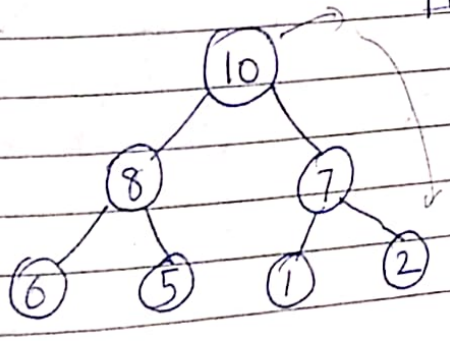
	Find-MAX	Delete MAX	Insert	increase(node value)	decrease key
MAX HEAP	O(1)	O(log n)	O(log n)	O(log n)	O(log n)

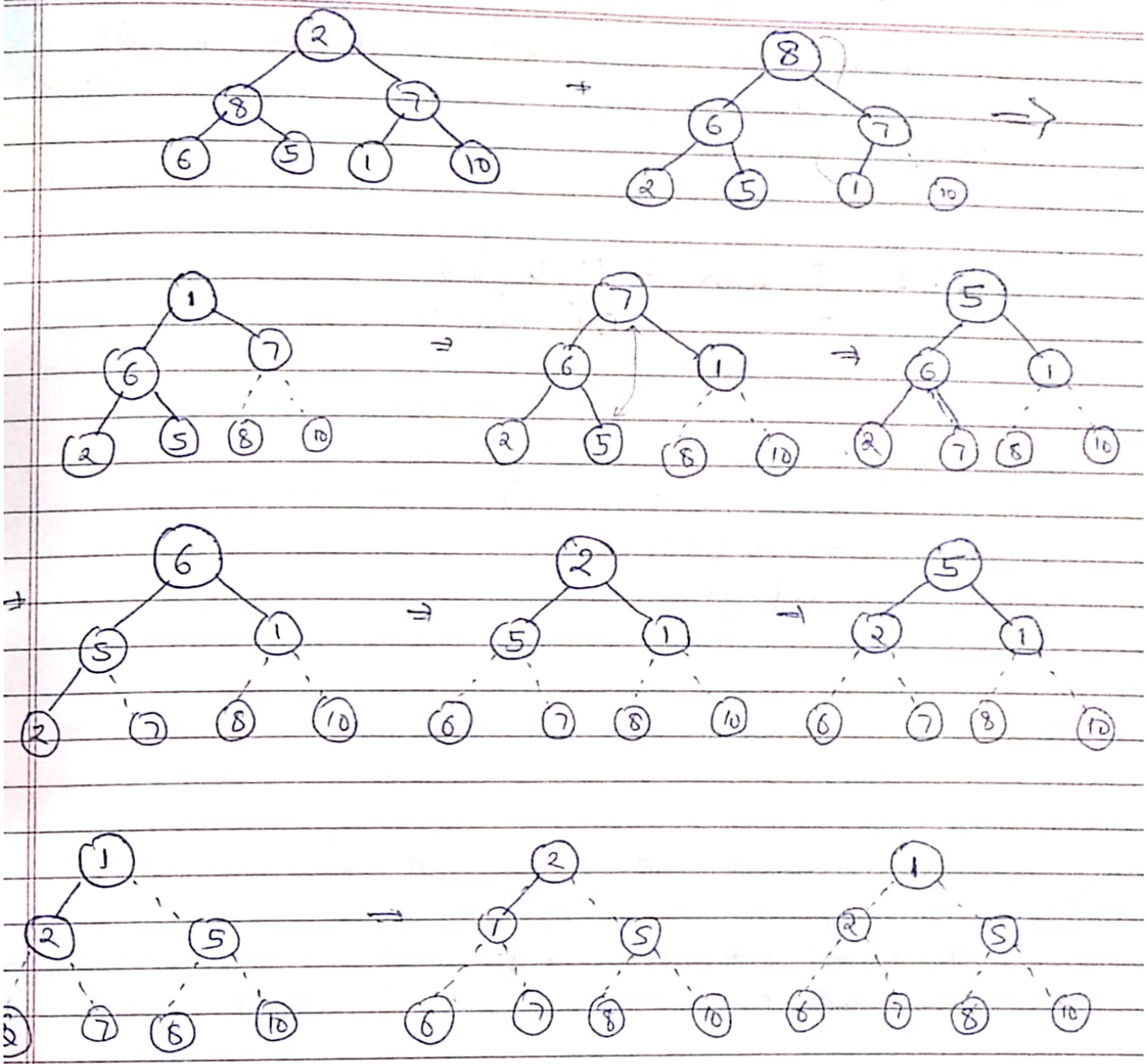
Find min	Search	delete
O(n)	O(n)	O(n+log n) O(n)

* Whenever we perform delete operation, the last node will replace that node & then we apply Heapify fn to arrange it.

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

10	8	7	6	5	1	2





1	2	5	6	7	8	10

Time complexity : $O(n \log n)$ → n element & they are calling Hepify, Heapify take logn time,

$\left(\frac{n}{2} \times \log n\right) \approx O(n \log n)$, Space complexity = $O(\log n)$

* In heap with 'n' elements with the smallest element at the root, the T^{th} smallest element can be found in.

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

delete 1^{st} min - $O(\log n)$

2^{nd} min - $O(\log n)$

6^{th} min - $O(\log n)$

7^{th} $\Rightarrow O(1)$

Insert - $6 \times O(\log n)$

$= O(\log n)$

* In a binary max heap containing n numbers, the smallest element can be found in time.

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

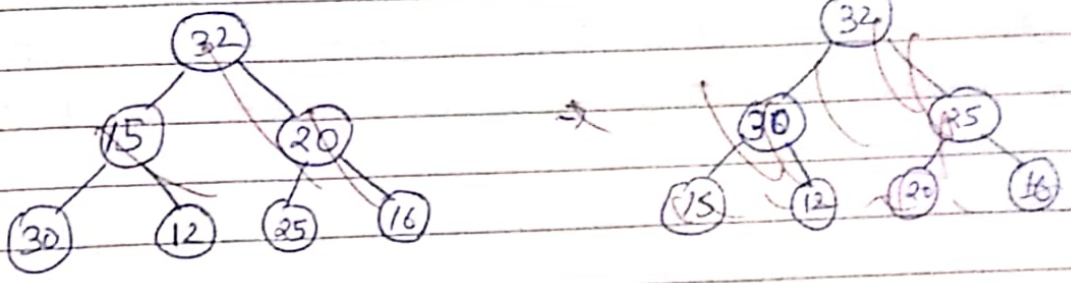
Smallest element is present at bottom, so we have to do $\left\lceil \frac{n}{2} \right\rceil$ comparisons

$O\left(\frac{n}{2}\right) \rightarrow O(n)$

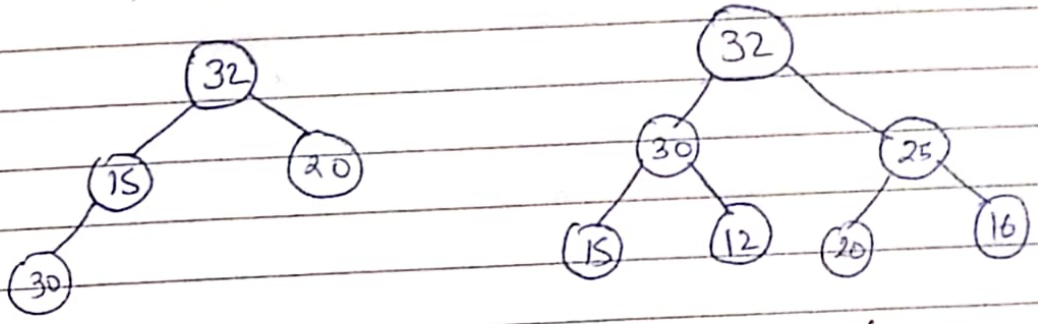
(1) (2) (3) (4)

(3)

32, 15, 20, 30, 12, 25, 16



While inserting any element we have to do checking.

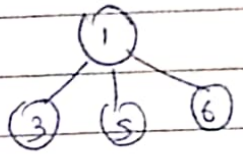


32 30 25 15 12 2016 (MAX Heap)

Max

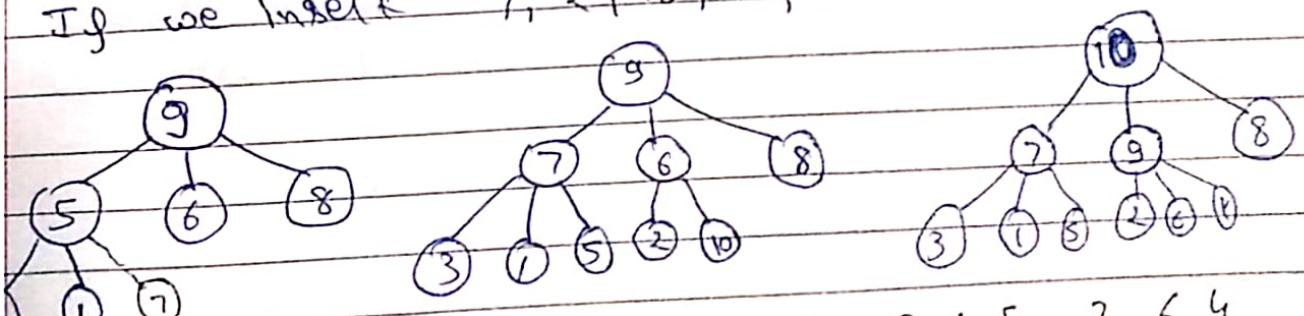
Which of the following is ternary Heap

1, 3, 5, 6, 8, 9
9, 6, 3, 1, 8, 5
9, 3, 6, 8, 5, 1
9, 5, 6, 8, 3, 1



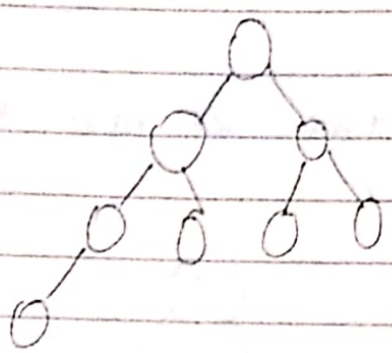
N.P
root is smaller.

If we insert 7, 2, 10, 4, then what will be resulted.



10, 7, 9, 8, 3, 1, 5, 2, 6, 4

* Consider the process of inserting an element into a max heap. If we perform a binary search on the path from new leaf to root to find the position of newly inserted element, the no of comparison performed are.



$O(\log n)$

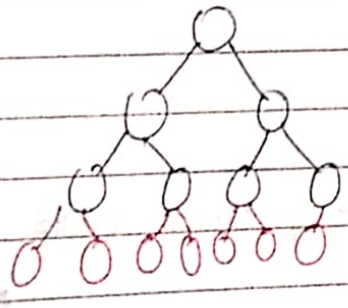
↓

number of element from leaf to root.

Then apply binary search : $O(\log(\log n))$

* We have a binary heap on n elements & wish to insert 'n' more element (not necessarily one after another) into this heap. The worst time required for this is.

a) $\Theta(\log n)$ ~~b) $\Theta(n)$~~ c) $\Theta(n \log n)$ d) $\Theta(n^2)$



Consider it as n element are present & now we have to make it heap which will be $O(n)$

RADIX SORT

Radix_Sort (A, d)

// Each Key in A[1..n] is a d digit integer.
// Digit are numbered 1 to d from right to left

for i=1 to d do
use a stable sorting algorithm to sort A on digit i.

8 0 4	0 0 1	0 0 1	0 0 1
0 2 6	0 5 2	8 0 4	0 0 5
0 0 5	8 0 4	0 0 5	0 2 6
0 6 4	0 6 4	0 2 6	0 5 2
0 5 2	0 0 5	0 5 2	0 6 4
0 0 1	0 2 6	0 6 4	8 0 4

Time Complexity: $\Theta(D(n+b))$

$\log_b l\,(D(n+b))$

l = largest no
b = base of no

if $l = n^k$

$\log_b n^k\,(D(n+b))$

$O(n \log_b n)$

Space Complexity: $O(b)$

```
Bubble Sort (int a[], n) {

int swapped i, j ;
for (i=0; i<n ; i++) {
 swapped = 0 ;

for (j = 0 ; j<n-i-1 ; j++) {
      if (a[j] > a[j+1]) {
   swap (a[j], a[j+1]);
    swapped = 1;
 }
}
  if (swapped == 0) :
       break;
}
}
```

Time complexity : $O(n^2)$
Space complexity : $O(1)$

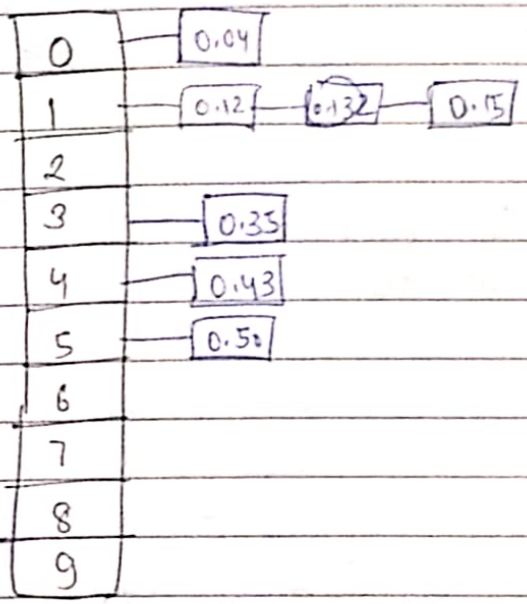
In best case, time complexity = $\Omega(n)$

Sorting

Bucket Algorithm

Sort a large set of floating point numbers which are in the range 0.0 to 1.0 & are uniformly distributed across the range?

0.85, 0.12, 0.43, 0.15, 0.04, 0.50, 0.132



In best case time complexity : $O(n)$

In worst case time complexity will be: All the no' come in one bucket, then again we have to n comparison: $O(n^2)$

Space complexity: $O(n+k)$

↙ To put all no' ↓ To put (0-9)

Counting Sort: It is a method with best time complexity for smaller range no' but it consume more space

(1-5) range

2 2 3 4 1 5 1 5

1	12
2	22
3	1
4	1
5	12

Output: 1 1 2 2 3 3 5 5

Time complexity: $O(n+k) = O(n+k)$

Space complexity: $O(k)$

(16)

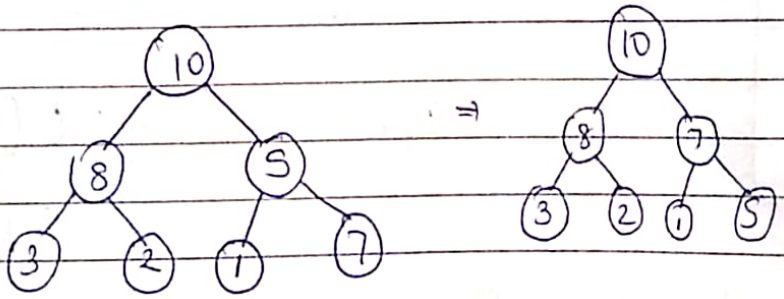
* The worst case running time of Insertion, Merge & quick sort are

A) $\theta(n^2)$, $\theta(n \log n)$, $\theta(n^2)$

* A priority queue is implemented as a max heap initially, it has 5 elements. The level order traversal of the heap is 10, 8, 5, 3, 2. The new element 1 & 7 are inserted into the heap in that order. The level order traversal of the heap after insertion of elements.

a) 10, 8, 7, 3, 2, 1, 5 b) 10, 8, 7, 2, 3, 1, 5

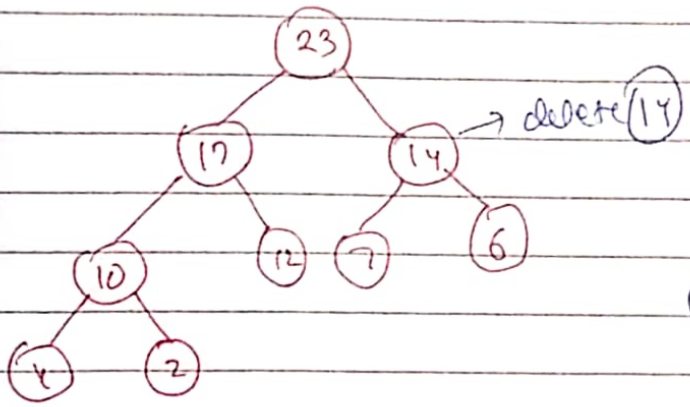
c) 10, 8, 7, 1, 2, 3, 5 d) 10, 8, 7, 5, 3, 2, 1

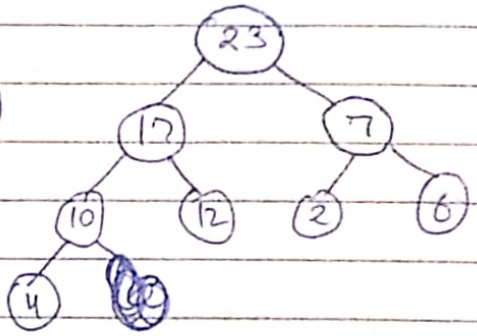


* An operator delete(i) for a binary heap data structure is to be designed to delete the item in the i^{th} node. Assume that the heap is implemented in an array S. refers to the i^{th} index of the array. If the heap has depth 'd' (no of edges on the path from the root to the farthest leaf), then what is the time complexity to refix the heap efficiently after the removal of the element.

3⇒ a) $O(1)$ b) $O(d)$ but not $O(1)$ c) $O(2^d)$ but not $O(d)$

d) $O(d\,2^d)$ but not $O(2^d)$

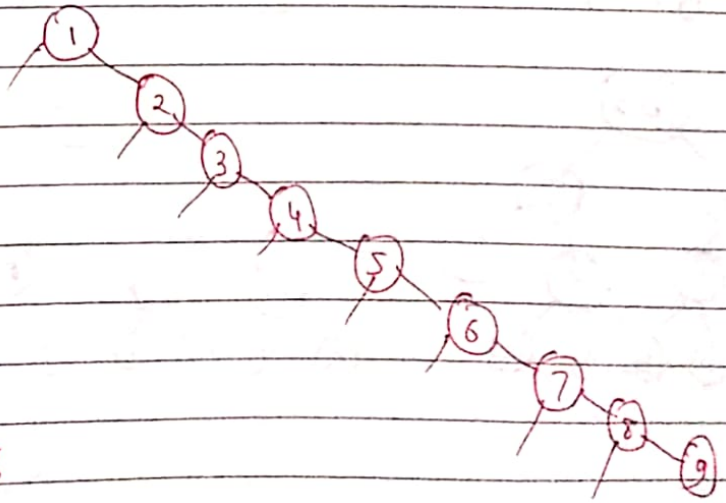




$O(\log n)$ time we can fix it. MAX-HEAPIFY algo is used. So $O(d)$

* A complete binary min-heap is made by including each integer in $[1, 1023]$ exactly once. The depth of a node in the heap is the length of the path from the root of the heap to that node. Thus the root is at depth 0. The max depth at which int '9' can appear is.

Total depth: $\log[1023] = 9$ depth.



Ans = 8

* Assume that the algo consider here sort the input sequence in ascending order. If the input is already in ascending order, which of the following is true?

I. Quick sort runs in $\theta(n^2)$ time

II. Bubble sort runs in $\theta(n^2)$ time

III. Merge sort runs in $\theta(n)$ time.

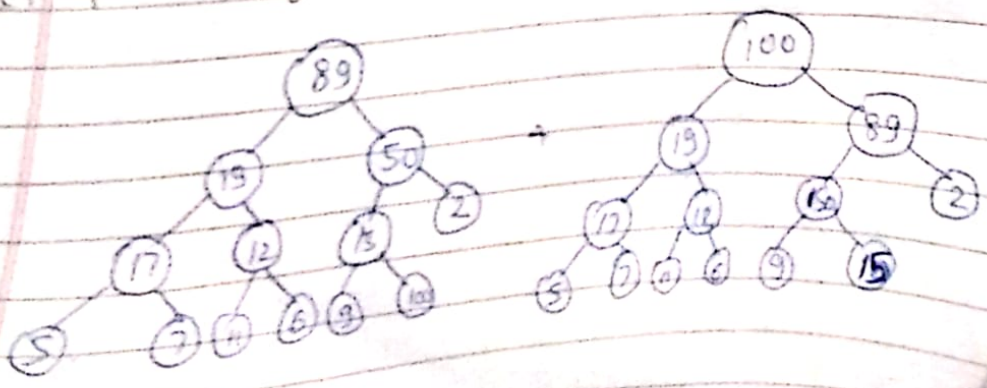
IV. Insertion sort runs in $\theta(n)$ time.

a) I & IV b) I & III

c) II & IV d) I & IV

* Consider the following array of elements $\langle 89, 19, 50, 10, 12, 15, 2, 5, 7, 11, 6, 9, 100 \rangle$. The min no of interchanges needed to convert it into a max heap is.

a) 4 b) 5 c) 2 d) 3



* Consider a complete binary tree where the left & right subtree of the root are max-heaps. The lower bound for the no of operation to convert the tree to a heap is

a) $\Omega(\log n)$ b) $\Omega(n)$ c) $\Omega(n \log n)$ d) $\Omega(n^2)$

Selection Sort

64 20 18 17 6

6 20 18 17 64

6 17 18 20 64

6 17 18 20 64

Time complexity : $O(n^2)$
space : $O(1)$
no of swap : $O(n)$

Search Algorithm

* In computer science, a search algorithm is an algorithm that retrieves information stored within some data structure.

* Data structure can include linked lists, array, search trees, hash table or various other storage methods.

* Searching also encompasses algorithm that query the database, such as the SQL SELECT command.

There are 2 method:

1. Linear Search
2. Binary Search

Linear Search: Linear search or sequential search is a method for finding a target value within a list.

How it works:

* It sequentially checks each elements of the list for the target value until a match is found or until all the elements have been searched

* It works on both sorted & unsorted data

Problem statement: Given a list L of n elements with values or records $L_0 \cdots L_{n-1}$, & target value T, the following subroutine uses linear search to find the index of the target T in L

Basic Algo:
Set i to 0.
If $L_i = T$, the search terminates successfully, return i.
Increase i by 1
If i go to step 2. Otherwise the search terminates unsuccessfully.

Implementation of Linear search algorithm in array:

```
int linearSearch (int *arr, int size, int target)
{
    for (int index = 0; index < size; index++)
        { if (arr[index] == target)
            return index; }
    return -1;
}
```

Time complexity
Best case: O(1)
Worst Case: $O(n+1) \Rightarrow O(n)$
Average case: $\frac{n+1+1}{2} = \frac{n+2}{2} = O(n)$

Reccurance relation:
$T(n) = T(n-1) + 1$ if $n > 1$
$T(1) = 1$
So we can say $T(n) = O(n)$

Implementation of Linear Search in Linked lists.

```
Struct node * linear_search_LinkedList (Struct node * head,
                                        int target)
{
 if (head) {
     while(head) {
         if (head -> data == target)
             return head;
         head = head -> next;
     }
 }
 return head;
}
```

Binary Search:

Binary search, also known as half-interval search or logarithmic search, is a search algorithm that finds the position of a target value within a sorted array. It does not work on unsorted array.

How it works:

- Binary search works on sorted array. It begin by comparing the middle element of the array with target value.
- If target value matches the middle element,

its position in the array is returned.

If the target value is less than the middle element then search continues in the left half of the array otherwise search continues in the right half of the array.

Implementation of Binary Search Algo in Array

```
int binarySearch (int arr[], int l_index, int r_index,
                  int target)
{
int m_index;
while (l_index <= r_index)
{
  m_index = l_index + (r_index - l_index)/2;
  if (arr[m_index] == target)
      return m_index;
  if (arr[m_index] < target)
      l_index = m_index + 1;
  else
      r_index = m_index - 1;
}
return -1;
}
```

Time complexity:

Recurrence relation: $T(n) = T(n/2) + 1$ if $n > 1$

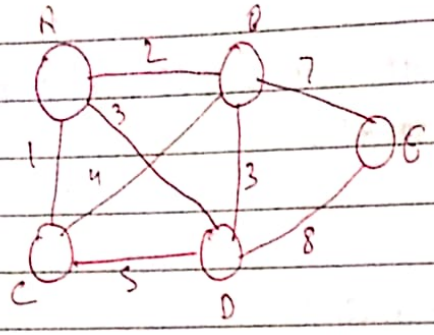
$T(n) = 1$ if $n = 1$

So, $T(n) = O(\log_2 n)$

Space complexity = $O(1)$

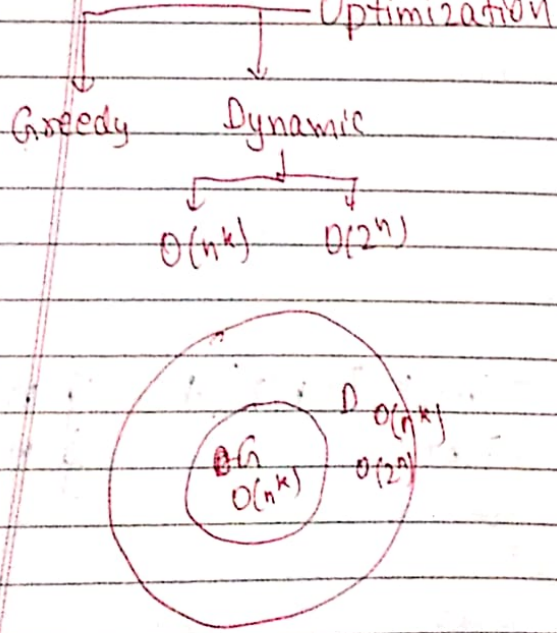
Optimization : By doing the optimization we can get

* Min Cost
* Max profit
* Max reliability
* Min risk



The shortest distance b/w any two node = $O(2^n)$
we will find all the cases & then select one with minimum path.

* But is not a good approach, it is time consuming procedure.



Greedy Algorithm: It is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious & immediate benefit. So the problem where choosing locally optimal also leads to global solution are fit for greedy.

Knapsack problem: We have to fill a bag by weight in such a way that we get max profit.

n = 3 , M = 20 unit

	Ob1	Ob2	Ob3
Profit	25	24	15
Weight	18	15	10
[illegible]			

Greedy about profit: First we will fill those weight who is having heighest profit.

	W	P
Ob1	18	25
Ob2	2	(24)(2/15)
	20	28.2

- Greedy about weight: So we have to fill maximum object in bag, so we start from less weight.

Obj 3	10	15
Obj 2	10	6
	20	31 profit

Now both the above method are not looking like a stable soln, because if we get any other instance, there there might be change in soln.

So best method is Profit/weight.

	Ob 1	Ob 2	Ob 3
Profit	25	24	15
weight	18	15	10
P/w	1.4	1.6	1.5

	w	p
Obj 2	15	24
Obj 1	5	7.5
	20	31.5

5/18

Hence we get the max profit from here.

Sor

Greedy Knapsack

```
{
for i=1 to n;
    Compute pi/wi;                    — O(n)
Sort object in non increasing order of p/w
                                      ≈ O(n log n)
for i=1 to n                          — O(n)
    if (m>0 && wi≤m)
        M = M-wi;
        P = P+pi;
    else break;

if (M>0)                              — O(1)
    P = P+ pi(M/wi);
}
```

M = 15 , n = 7

Object	1	2	3	4	5	6	7
Profits	10	5	15	7	6	18	3
Weights	2	3	5	7	1	4	1
	5	1.6	3	1	6	4.5	3

Sorting them: (5, 1, 6, 3, 7, 2, 4)

M = ~~15~~ ~~14~~ ~~12~~ ~~8~~ ~~3~~ 3 2 0

P = 6 + 10 + 18 + ~~10~~ 15 + 3 + ~~3~~ 3.3 = 55.3

Time complexity: $O(n) + O(n \log n) + O(n) + O(1)$
$= O(n \log n)$

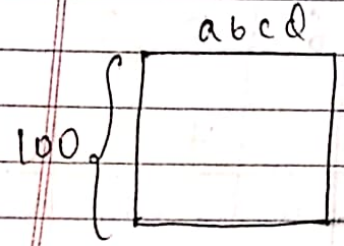
Space complexity: $O(n)$

HUFFMAN CODING:

It is used to compress a file. To decrease the size of file.

* Here we see the occurance of character, the character with highest occurance should allocate least bit, ~~so on~~ ~~so on~~.
* As compared to the heighest one, other characters should be less.

let us suppose that we are having a file, which uses 4 character type, a, b, c, d. And there are total 100 char. Each char take 2 bit, so size of file will be 200 bit.



a = 00
b = 01
c = 10
d = 11

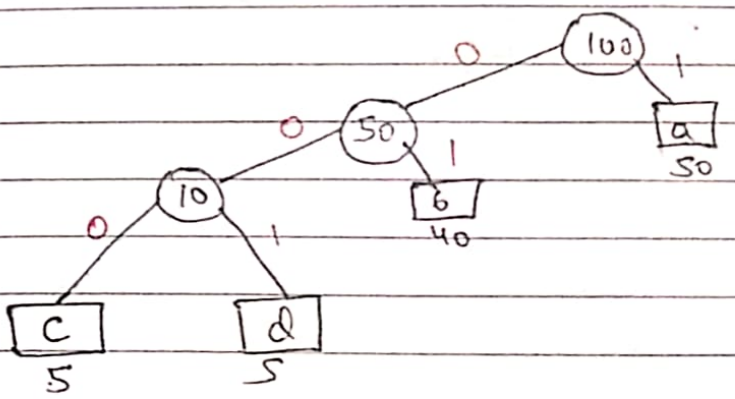
$100 \times 2 = 200$ bit

* Now solve the same using Huffman code:

a - 50	0	50 x 1
b - 40	10	40 x 2
c - 5	110	5 x 3
d - 5	111	5 x 3
		160 bits

The prefix of every char should be different, so that we can easily distinguish b/w the char & easily decode the code.

We put the minimum at bottom & then join them in binary way.



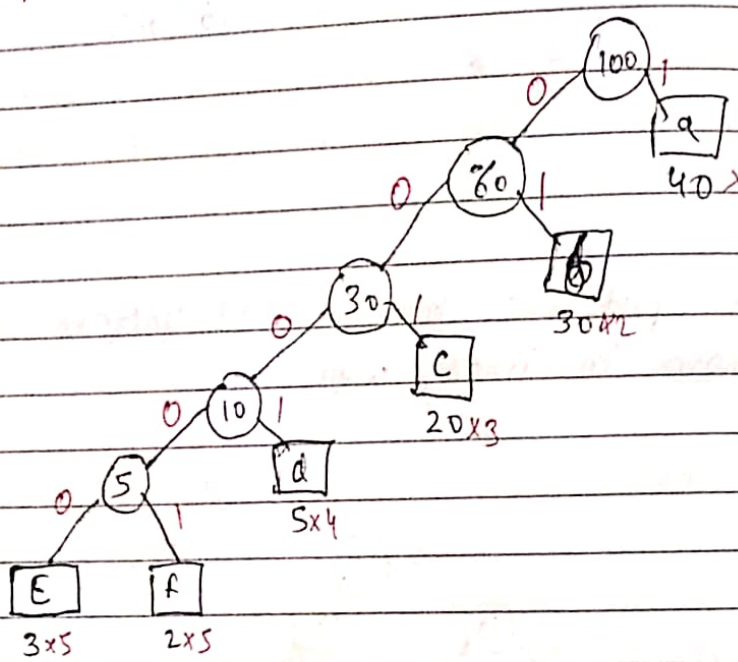
a = 1
b - 01
c - 000
d - 001

Total bit = 160

$\frac{\text{Bits}}{1 \text{ Char}} \Rightarrow \frac{160}{100} = 1.6 \text{ bits/Char}$

Time Complexity : $O(n) + O(n \log n) + O(n) + O(n)$
$= O(n \log n)$

a = 40 , b = 30 , c = 20 , d = 5 , e = 3 , f = 2



a - 1	- 40×1
b - 01	- 30×2
c - 001	- 20×3
d - 0001	- 4×5
e - 00001	- 5×2
f - 00000	- 5×3
	205 bits

Total word = 100.

$\frac{\text{Bits}}{\text{Char}} = \frac{205}{100} = 2.05$ bit/char

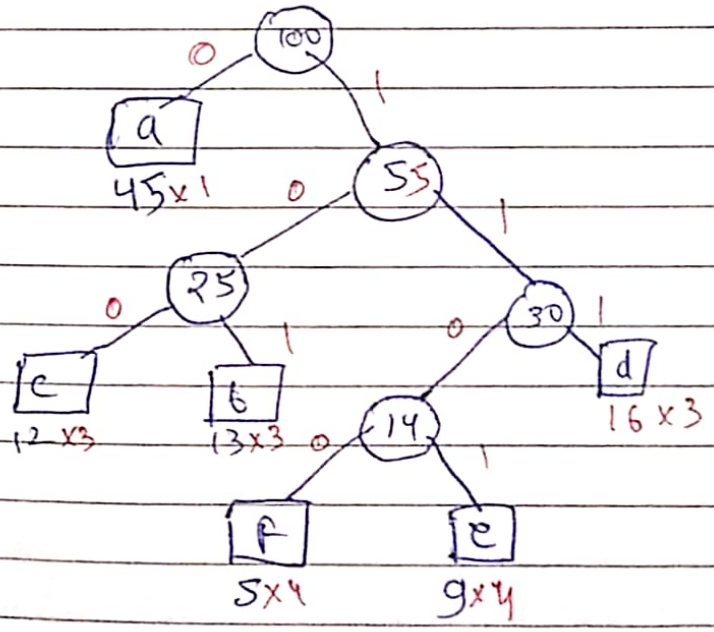
Without this : 100 × 3 = 300 bits used

HUFFMAN(C)

{ ——— → set of all character
n = |C|
Make a min heap 'Q' with C — O(n)

for i = 1 to n-1 — O(n)
(allocate a new node z)
z.left = x = Extract-min(Q) — O(log n)
z.right = y = Extract-min(Q) — O(log n)
z.freq = x.freq + y.freq
Insert(Q, z) — O(log n)
return (Extract-min(Q))
}

a	b	c	d	e	f
45	13	12	16	9	5



Important:

a - 0	= 45x1 = 45
b - 101	- 3x13 = 39
c - 100	-3x12 = 36
d - 111	-3x16 = 48
e - 1101	-4x9 = 36
f - 1100	-4x5 = 20
	224

Time complexity - $O(n) + O(n)(O(\log n) + O(\log n) + O(\log n))$

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

$= O(n \log n)$

Space complexity: $O(n)$ [for building a tree]

We don't have to use sorting algo, because it increase the time complexity while inserting & deleting, it consume $O(n^2)$

Suppose the letters a, b, c, d, e, f have probabilities 1/2, 1/4, 1/8, 1/16, 1/32, 1/32.

1. Which of the following is the huffman code for the letters a, b, c, d, e, f

a) 0, 10, 110, 1110, 11110, 11111

b) 11, 10, 011, 010, 001, 000

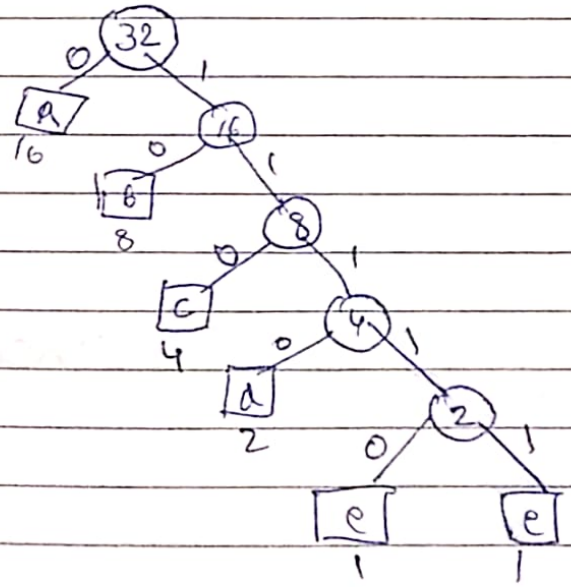
c) 11, 10, 01, 001, 0001, 0000

d) 110, 100, 010, 000, 001, 111

2. What is the avg length of the correct answer to question 1

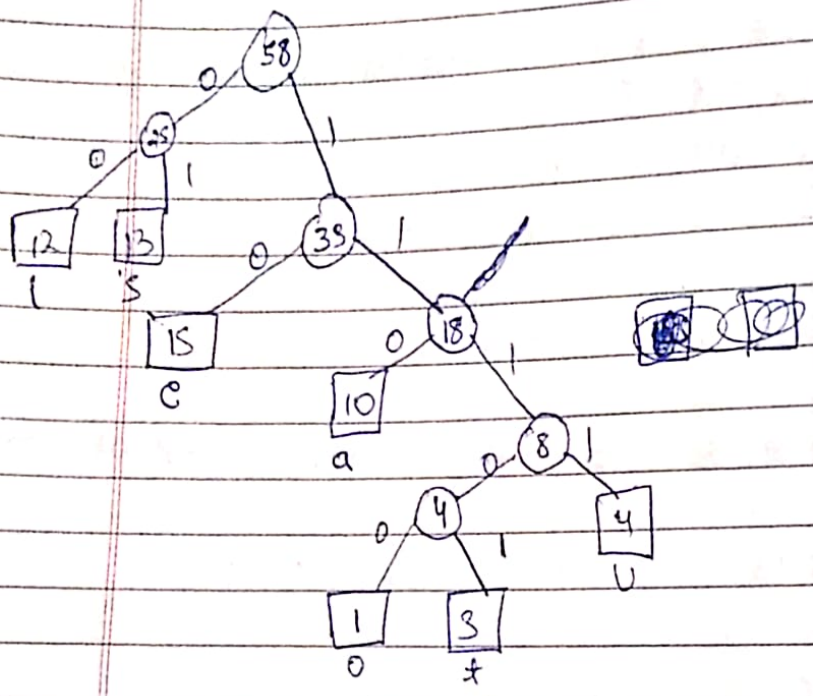
a) 3 b) 2.1875 c) 2.25 d) 1.9375

a	b	c	d	e	f
16	8	4	2	1	1



$$\frac{16\times1 + 8\times2 + 4\times3 + 2\times4 + 1\times5 + 1\times5}{32} \Rightarrow \frac{62}{32} = 1.9375$$

a	e	i	o	u	s	t
10	15	12	3	4	13	1



a - 110
e - 10
i - 00
o - 11100
u - 1111
s - 01
t - 11101

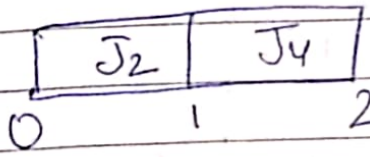
Job sequencing with deadlines.

		J_1	J_2	J_3	J_4
Deadline	D	2	1	1	2
Profit	P	6	8	5	10

Every job has a deadline to finish it, it take just 1 unit to finish it. In what order we finish the job so that profit will be maximum.

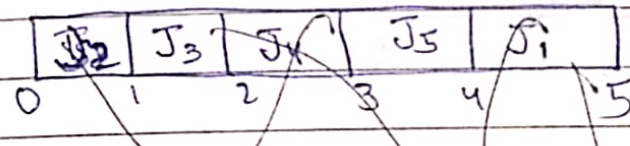
* first select the highest deadline & make it partition in one unit.

* Then select a job with highest profit put it one of the last block as possible so that it can finish itself.

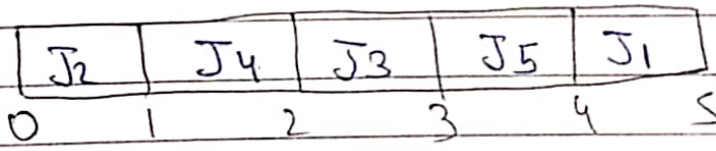


, so profit = 18

	J_1	J_2	J_3	J_4	J_5	J_6
D	5	3	3	2	4	2
P	200	180	190	300	120	100



180 + 190 + 300 + 120 + 200 = 990



180 + 300 + 190 + 120 + 200 = 990

Time Complexity:

Sord the job i $O(n \log n)$

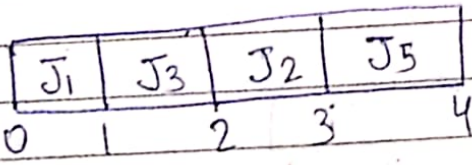
now putting them in proper place = $n \times O(n)$ → Find place
(no of element)

$= O(n^2)$

Jobs	J_1	J_2	J_3	J_4	J_5
Profits	2	4	3	1	10
Deadline	3	3	3	4	4

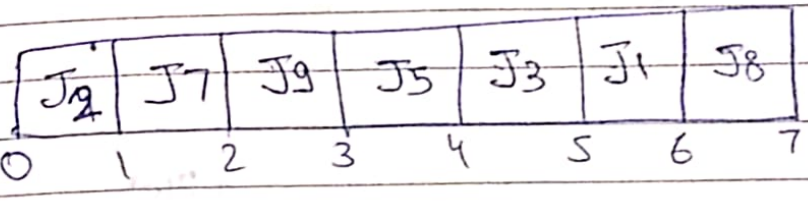
~~Sorted then J5, J1, J2, J3, J4.~~

Sort then J_5, J_2, J_3, J_1 J_4



2 + 3 + 4 + 10 = 19 profit

	J_1	J_2	J_3	J_4	J_5	J_6	J_7	J_8	J_9
P	15	20	30	18	18	10	23	16	25
D	7	2	5	3	4	5	2	7	3

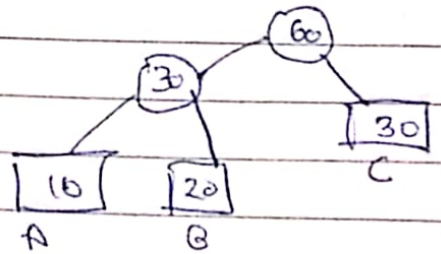


$20 + 23 + 25 + 18 + 30 + 15 + 16 = 147$

Optimal MERGE Pattern:

It is a sol^n to merge two or more file in such a way that there would be less no of moment of data record.

So first we merge those file which contain the less no of record & in this way we continue. It is similar to huffman code.

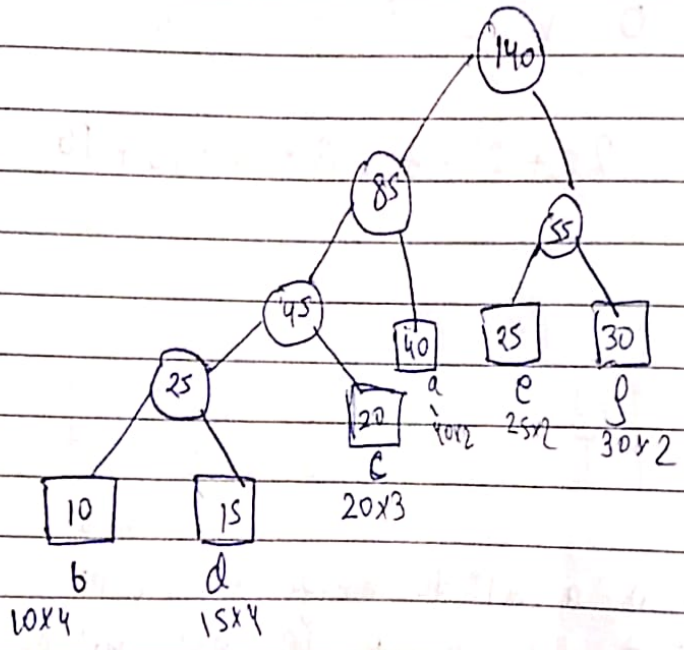


So total moment =
$10 \times 2 + 20 \times 2 + 30$
$= 90$
or $30 + 60 = 90$

Time complexity = O(n log n)

Min heap will be used, code is similar to Huffman code.

a	b	c	d	e	f
40	10	20	15	25	30

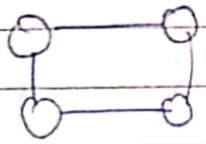


25 + 45 + 85 + 140 + 55 = 350 record movement

GRAPHS:

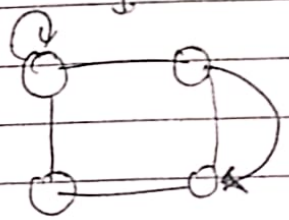
(V, E)

1. Simple graph



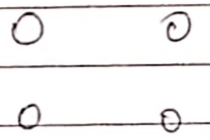
B/w any two vertex there is atmost one edge.

Multi graph



If b/w any two vertex, there is atleast 2 edges.

Null graph



No edges

Maximum no' of Edges = ${}^{n}C_2$ (n = no' of vertex)

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

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

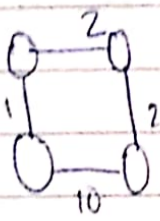
$$E = O(V^2)$$

Apply log on both side. // not confirmed.

$$V = \Theta(\log E)$$

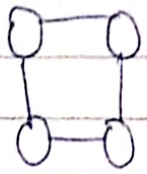
Spanning tree: It is a tree which connect all the node in such a way that weight would be minimum. It doesn't form any cycle

(i) Weighted graph:



20 · weight is given,

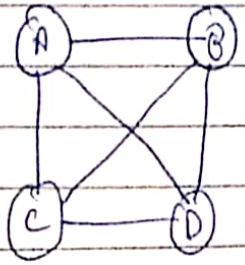
(ii) Unweighted graph:



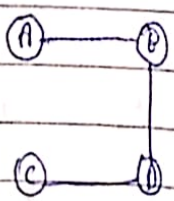
Minimum no of edges a graph can contain so that all the vertex are connected.

n = no of vertex.

'n-1' no of edges, this we have to find with the help of spanning tree,



MAX = 6

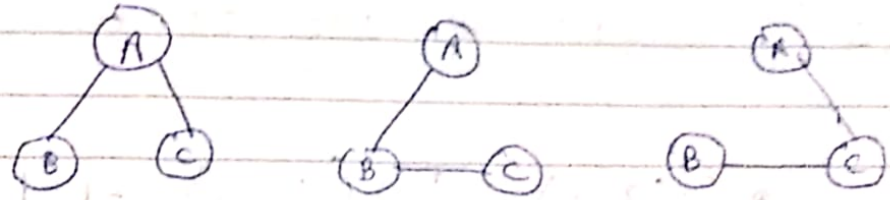


Min = 3

(1)
(2)
(3)

* How many different graph can be created with minimum no of edges, for complete graph is given.

$n = 3$



no of different possible graph = 3

$n = 4$

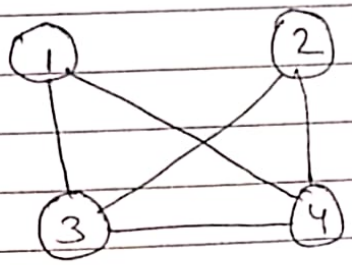
no of possible = 16

Formula : (n^{n-2})

Sometime it give the graph, & tell how many different graph are possible with minimum edge.

Kirchoff matrix :

1) Diagonal 0's ⇒ Degree of node
2) Non diagonal 1's → "-1"
3) No diagonal 0's → '0'



	1	2	3	4
1	0	0	1	1
2	0	0	1	1
3	1	1	0	1
4	1	1	1	0

⇒

	1	2	3	4
1	2	0	-1	-1
2	0	2	-1	-1
3	-1	-1	3	-1
4	-1	-1	-1	3

diagonal

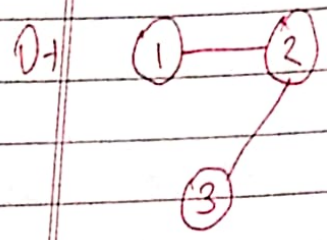
→ find determinant

ST = Cofact of any element

$= 2\left(2(9-1) - (-1)(-3-1) - 1(1+3)\right)$

$= 8$

Hence 8 graph are possible



	1	2	3
1	0	1	0
2	1	0	1
3	0	1	0

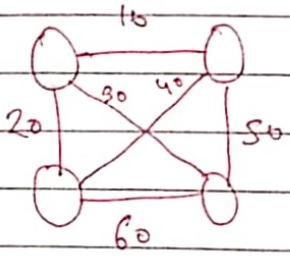
+

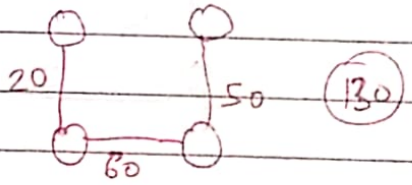
	1	2	3
1	1	-1	0
2	-1	2	-1
3	0	-1	1

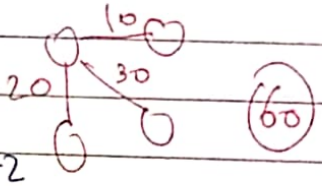
$= 2 \div (1) = 1$ graph possible

Prims: Algorithm:

(1) first choose an edge with minimum length.
(2) start exploring the neighbor of connected vertices.
(3) If the vertex, which ~~started~~ have the minimum length from the connected vertex will be choosen & there is a condition that the vertex which we are going to connect is not connected already in graph.
(4) A graph can has more than one spanning tree.

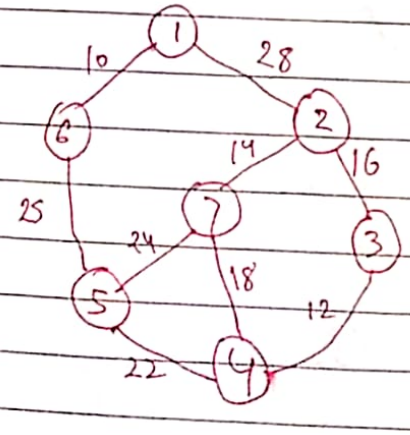




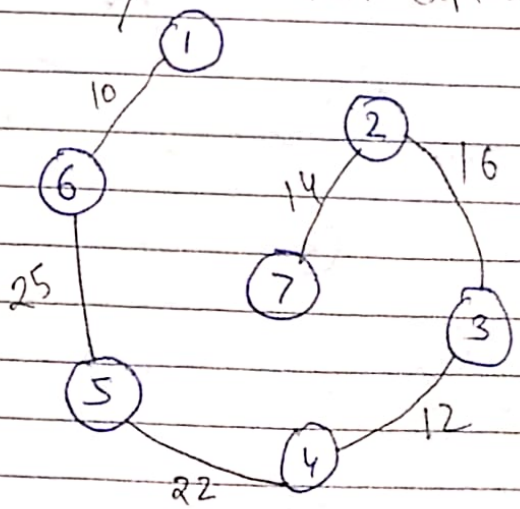


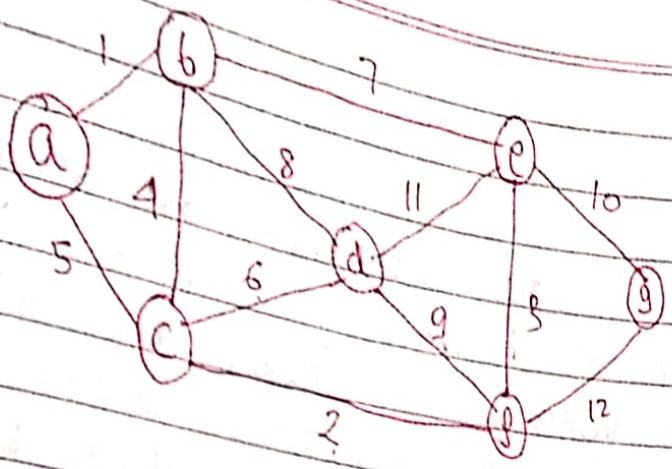
Total possibility = n^{n-2}
$= 4^{4-2} = 16$

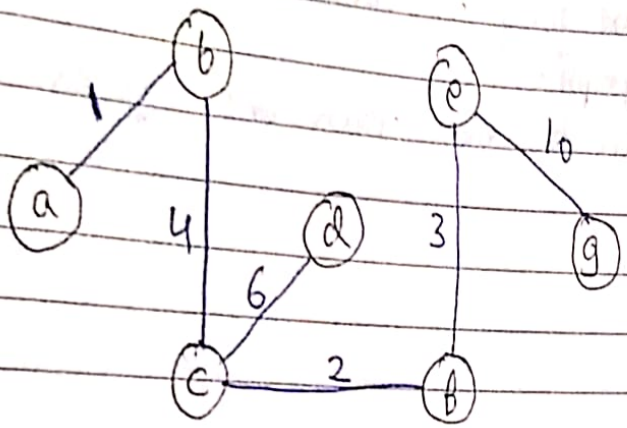
(1)



Start from smallest edge & then explore.

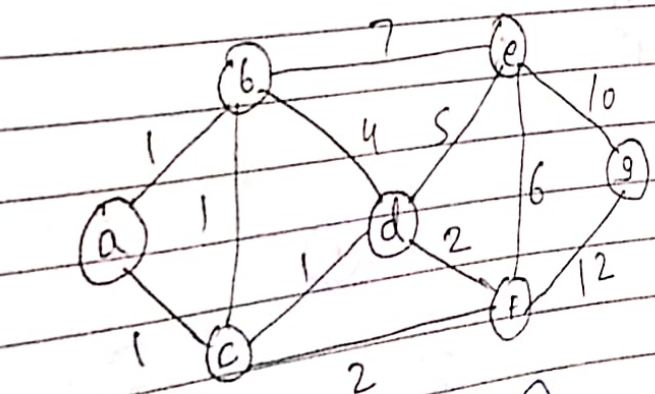


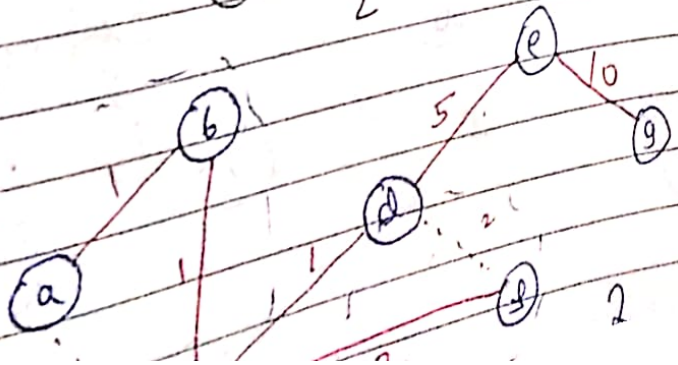




= 26

* Find the total no of minimum spanning tree possible



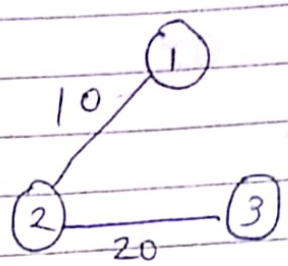


So total = 3 x 2 = 6 possible case.

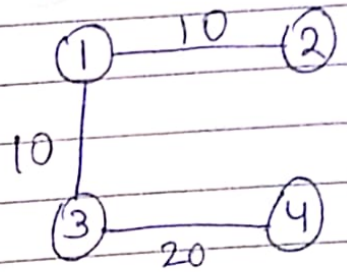
Minimum spanning tree = 20

: Find the minimal spanning tree.

	1	2	3
1	0	10	30
2	10	0	20
3	30	20	0

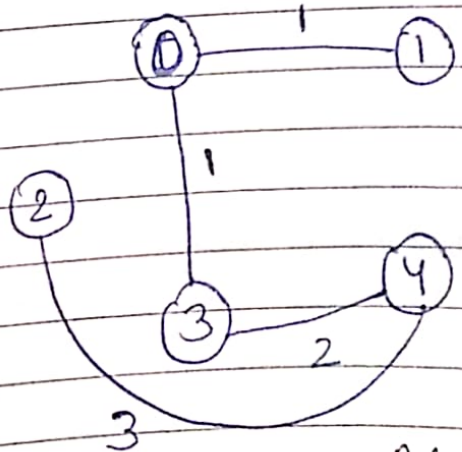


	1	2	3	4
1	0	10	10	50
2	10	0	40	30
3	10	40	0	20
4	50	30	20	0



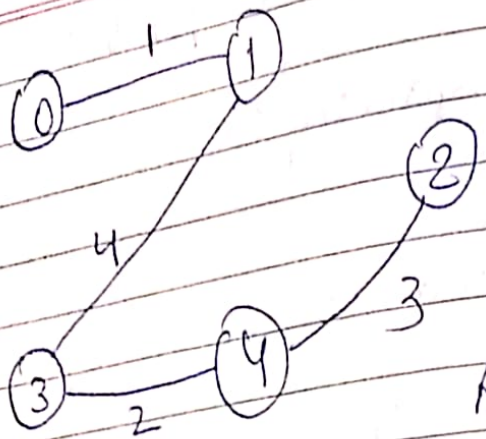
ans = 40

	0	1	2	3	4
0	0	1	8	1	4
1	1	0	12	4	9
2	8	12	0	7	3
3	1	4	7	0	2
4	4	9	3	2	0



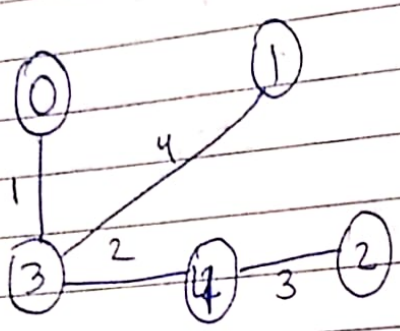
Ans 7

There is change in question, if node 0 has degree only 1, then find minimal.



All = 10

Or



All 10

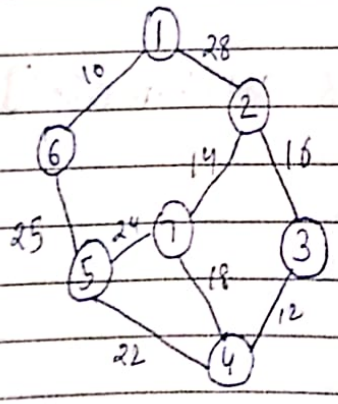
Algorithm Prim (E, cost, n, t)

// E is the set of edges. Cost is (n x n) adjacency matrix.
// MST is computed & stored in array t[1:n-1, 1:2]
;
1 let (K, l) be an edge of min cost in E;
  Min cost = cost[K, l];
  t[1,1] = K ; t[1,2] = l;
4 for (i = 1 to n)
5 if (cost[i, l] < cost[i, K]) then near[i] = l;
6 else near[i] = K;
near[K] = near[l] = 0
8 for (i = 2 to n-1) {
9 let j be an index such that near[j] ≠ 0 And
10 cost[j, near[j]] is minimum;
11 t[i,1] = j ; t[i,2] = near[j];
12 minCost = minCost + cost[j, near[j]];
13 near[j] = 0
14 for K = 1 to n do
15 if (near[K] ≠ 0 and (cost[K, near[K]] > cost[K, j])
    then near[K] = j;
}
}

Time Complexity : $O(n^2)$, If we use heap we get $O(E \log n)$ which is better

* MST-PRIM (G, Cost, r) // using min heap

```
{
1  For each vertex U ∈ G vertices
   {
2    U.key = ∞                    } O(V)
3    U.π = NIL                    }
   }
4  r.key = 0
5  Q = G.vertices                 - O(V)
6  while(Q ≠ ∅)                   - O(V)
   {
7    U = Extract-MIN(Q)           - O(log V)
8    For each vertex 'v' adjacent to 'U'  --> O(V)
9    { If v ∈ Q & cost(U,v) < v.key
10     v.parent = U;
11     v.key w(u,v).              - O(log V)
   }
```



	1	2	3	4	5	6	7
Key	∞0	∞28	∞	∞	∞	∞6	∞
π	N,	N1	N	N	N	N1	N

Parent

Similarly solve it.

$O(v^2 \log v)$ But it is possible in worst case

Extract min - $O(v \log v)$
Decreased-key - $O(E \log V)$
Build-heap - $O(E)$

$T(v) = O(v \log v + E \log v + E) = O(E \log v)$ // min heap

$= O(v \log v + E)$ // Fibonacci heap

$= O(v^2)$ // No heap

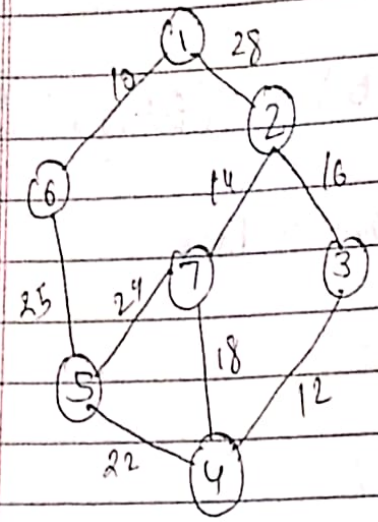
$O(E \log v) \iff O(v^2)$

Dense graph : More edges $E = O(v^2)$
$O(v^2 \log) - O(v^2)$

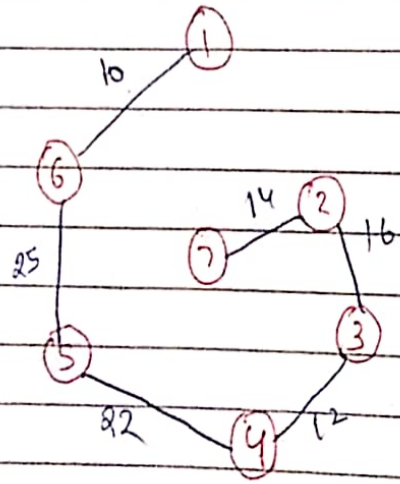
Sparse graph : less edges $E = O(v)$

$O(v \log v) - O(v^2)$

Kruskal's Algo:



* Connect the vertices with least weight
* We have to take care of cycle, if it is formed, we will not do that step & move forward
* ~~Max~~ No. of edges would be 'V-1'.



O(ElogV)

Let 'G' be ~~the~~ an undirected connected graph with distinct edge weights. Let max be the edge with maximum weight & min be the edge with minimum weight. which of the following is false.

Every minimum spanning tree of 'G' must contain min.

If max is in a minimum spanning tree, then its removal must disconnect 'G'

No minimum spanning tree contain 'max'.

'G' has a unique minimum spanning tree.

Let 'w' be the minimum weight among all weights in an undirected connected graph. Let 'e' be a specific edge of weight 'w'. Which of the following is false.

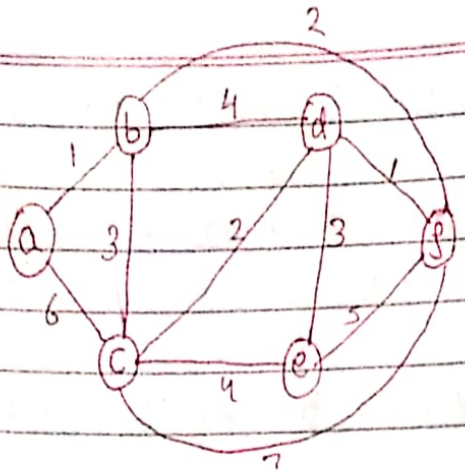
There is a minimum spanning tree containing 'e'.

If 'e' is not in a minimum spanning tree 'T', then in the cycle formed by adding 'e' to T', all edges have the same weight.

Every minimum spanning tree has an edge of weight 'w'

'e' is present in every minimum spanning tree.

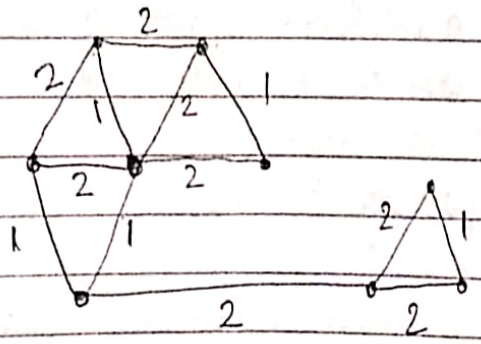
In this case, weight can be repeated, it is not mentioned that all weight are different.

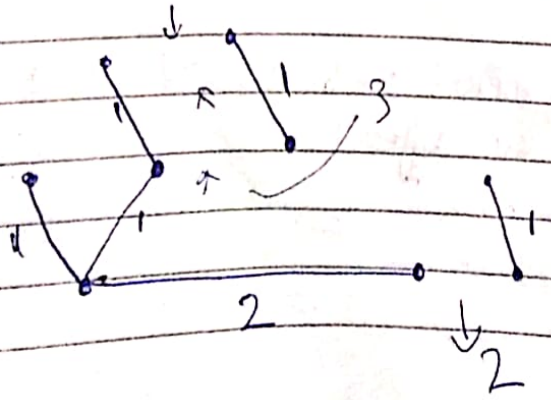


* Which of the following cannot be sequence of edges added in that order to minimum spanning tree using Kruskal's algorithm.

a) (a-b) (d-f) (b-f) (d-c) (d-e)
b) (a-b) (d-f) (d-c) (b-f) (d-e)
c) (d-f) (a-b) (d-c) (b-f) (d-e)
d) (d-f) (a-b) (b-f) (d-e) (d-c)

* The number of distinct minimum spanning trees for the weighted graph below is :

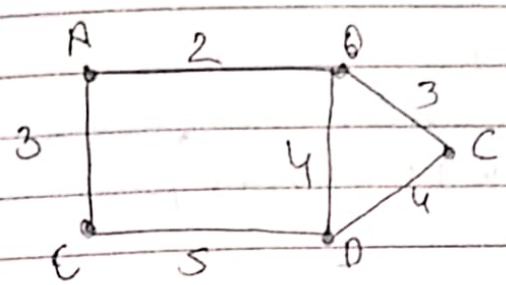


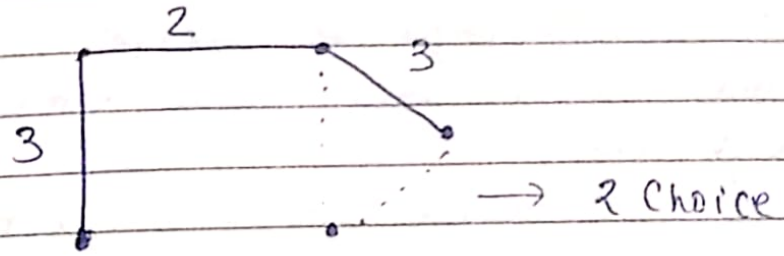


Complete graph : ~~From core~~ Every vertex has an edge to every other vertex.

So total = 3 x 2 = 6 spanning tree possible.

How many minimal spanning tree is possible

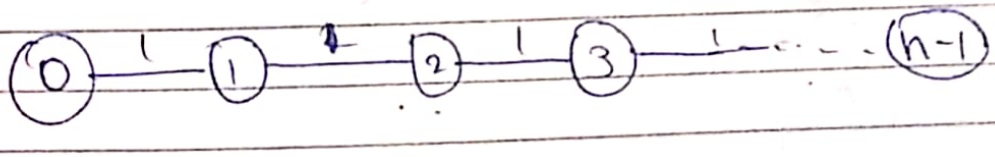


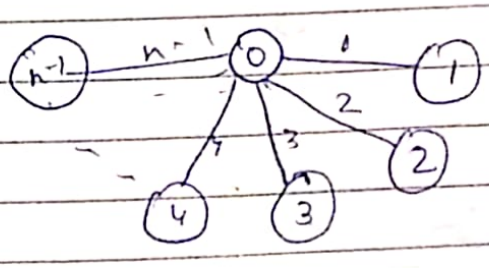


A complete, undirected, weighted graph 'G' is given on the vertex {0, 1, ... n-1} for any fixed 'n'. Draw the minimum spanning tree of G if

i) The weight of edge (u,v) is |u-v|

ii) The weight of the edge (u,v) is (u+v)





SHORTEST PATH:

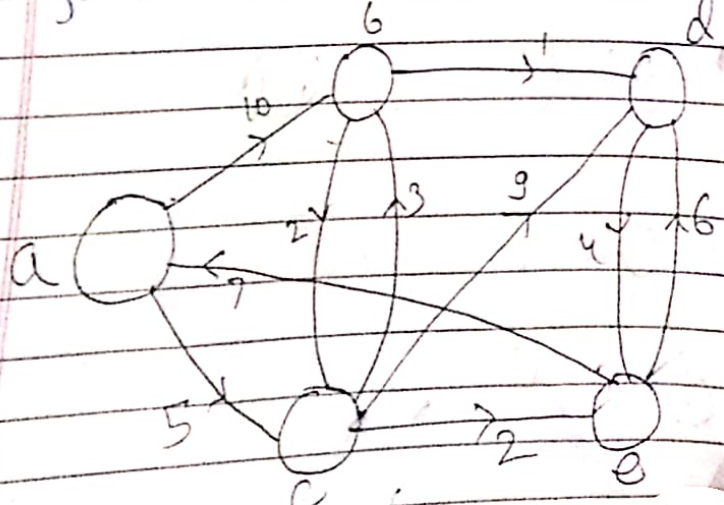
The shortest distance b/w source & all the vertices.

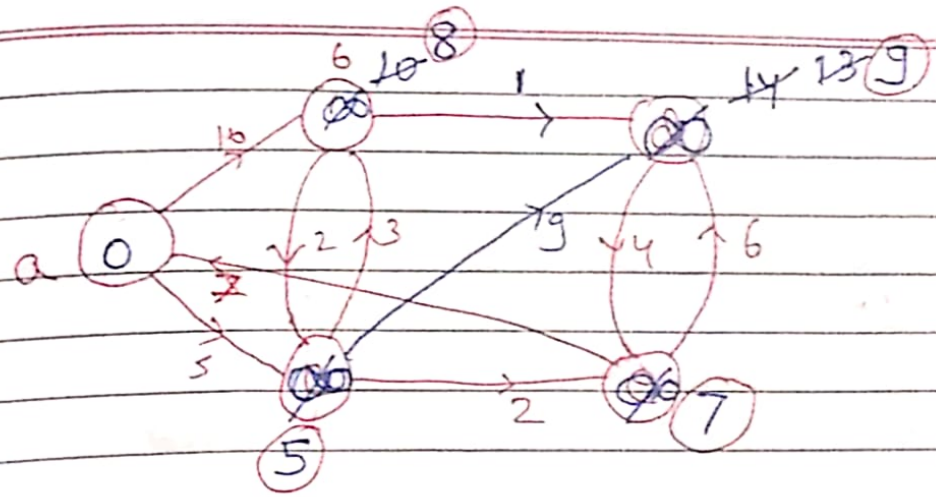
Relaxing an edge (v, u)

$$\left(\text{if } d(u) > d(v) + c(v,u) \right)$$
$$d(u) = d(v) + c(v,u)$$

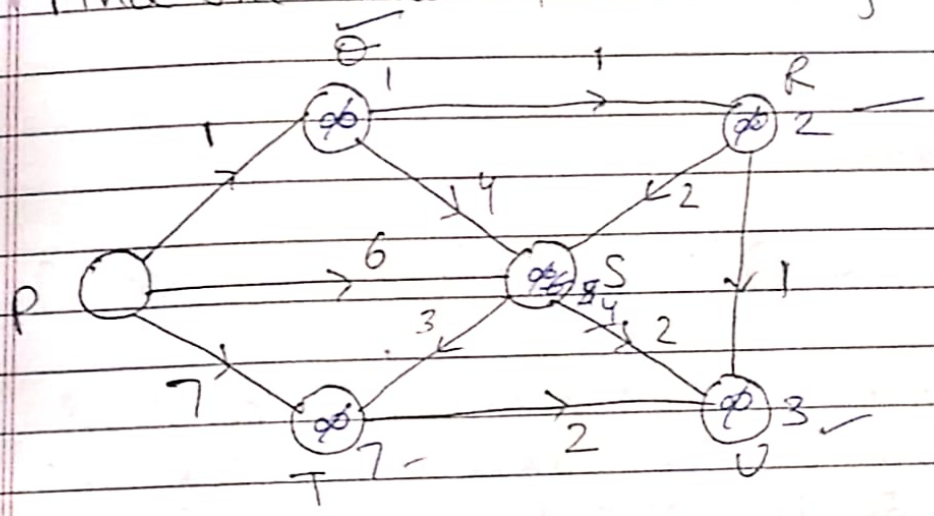
DIJKSTRA Algorithm:

* It is not applicable for negative weights.
* First select the source, then make every other vertex weight ∞.
* Now look at the vertex which is directly connected to source & fix its weight. if it is smaller than it's previous one.
* Then select a node with least weight, explore every vertex which is connected to it but don't check for the vertex which has already explored.
* In this way we get the shortest distance from the source.





Q- find the order in which we find the shortest path:



ORUST (Generally it is asking the shortest distance of source from all other node in ascending order)

Dijkstra (G, w, S) — O(V)

1. Initialize - Single - Source (G, S)
2. S = ∅
3. Q = G.V // Build heap — O(V)
4. While Q ≠ ∅
   u = Extract-min(Q)
   S = S ∪ {u}
   for each vertex v ∈ G.Adj[u]
      Relax (u, v, w)

Time complexity

Extract-min → O(log V) × O(n) = O(n log V)

Decrease-key · Relax : Decrease the value node S the arrange it, in worst case all the edges create the decrease, the heap has to take O(log V) to balance it & there are E edges which are causing it.

O(log V) × O(E) = O(E log V)

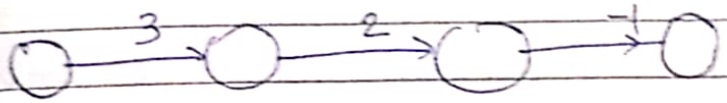
Total time complexity: O(V + V log V + E log V)

= O(E log V)

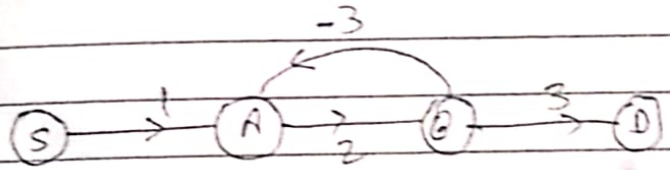
Problems arise while discovering the shortest path b/w vertices

If any node with 0 degree, i.e not connected to any vertex

If there is negative weight cycle.



Distance = 4, it is true because no negative cycle is present.



0	2	6
-1	1	5
-2	0	4

Create different result for every -ve cycle.

Algo

-ve cycle is present

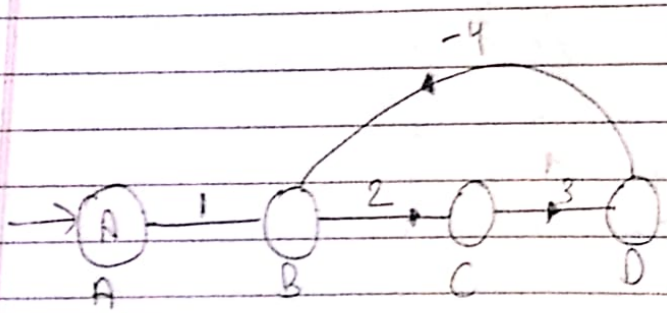
-ve cycle is not present

Our algo should be design in such a way that it can tell us the weather the -ve cycle is present or not.

Dijkstra fails to tell us this difference.

	A	B	C
1	0	∞	∞
2		10	∞
3			9

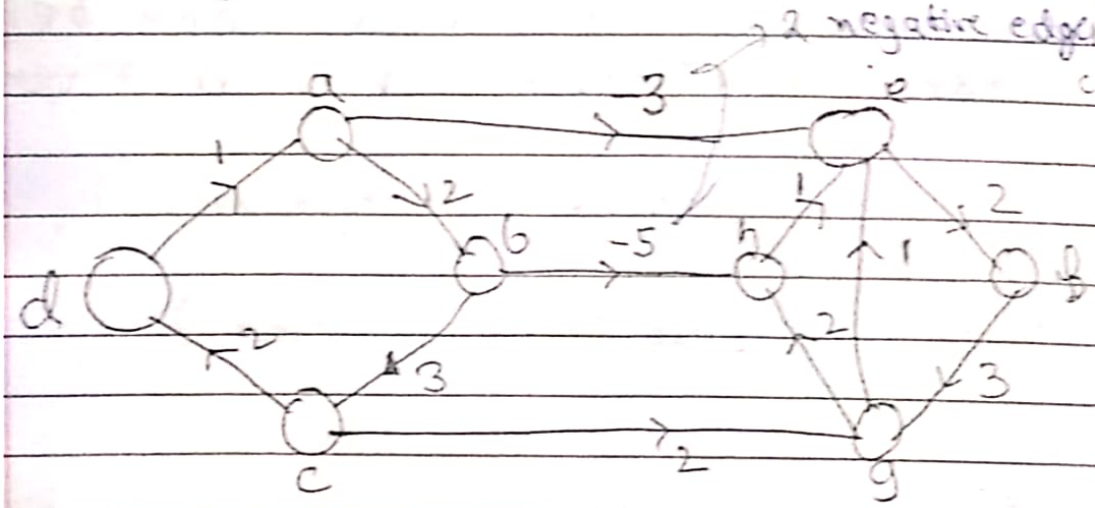
Whenever we deleted any node, we should confirmed that its weight value will not decrease after that. Dijkstra properly work on -ve edges wherever cycle is not formed.



	A	B	C	D
1	0	∞	∞	∞
		1	∞	∞
			3	∞
				6

Even cycle is present it is giving correct result, it only works for some example. If in place of -4, there is -6, it totally fails.

If cycle is formed, then there may be case that negative weight cycle doesn't work properly.
If cycle is not present it may works properly.



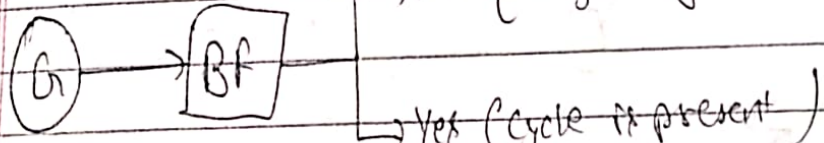
→ 2 negative edges in a is not permitted cycle. But there is 2 way to reach it without going through negative edges.

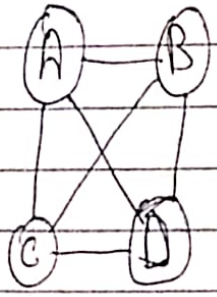
	a	b	c	d	e	f	g	h
	0	∞	∞	∞	∞	∞	∞	∞
a		2	∞	∞	-3	∞	∞	∞
e		2	∞	∞		-1	∞	∞
f		2	∞	∞			2	∞
g		2	∞	∞				4
b			5	∞				-3
h			5	∞				
c				7				
d								

I have always checked, whenever there is negative cycle.

Bellman Ford: This method can recognise any negative cycle present in a graph.

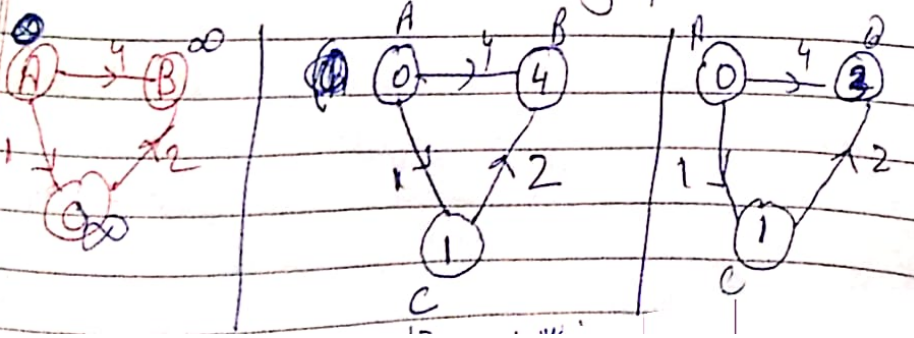
* The largest b/w any two node can't be greater than 'n-1' edges, where n is vertic

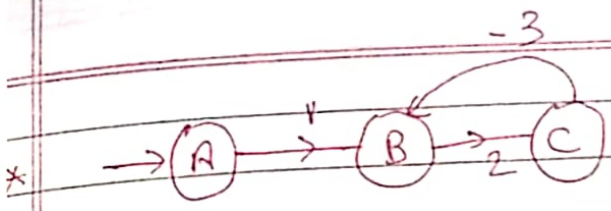


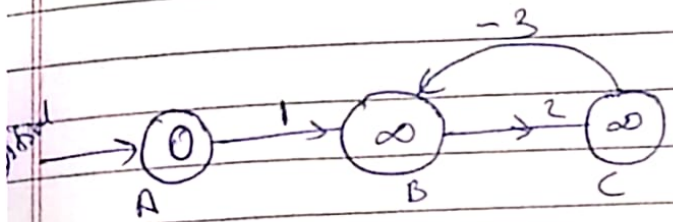


The longest path b/w A & D will be '3'. Condition, we can't access a vertex more than once.

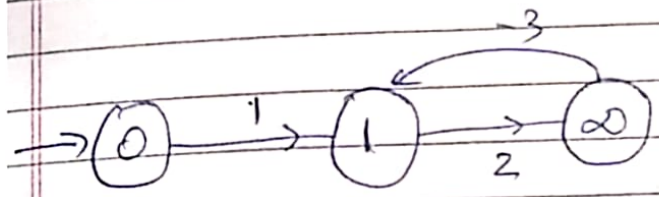
* If there are 'n' node, we have to 'n-1' relaxation.
* We will visit every node 'n-1' time.
* At last we do one more relaxation, if we again get a shorter path, it indicates that there is (-ve) edge cycle.
* It is a time consuming process.



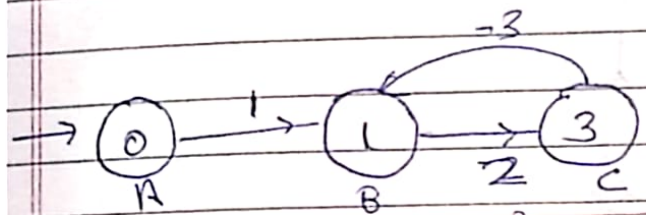




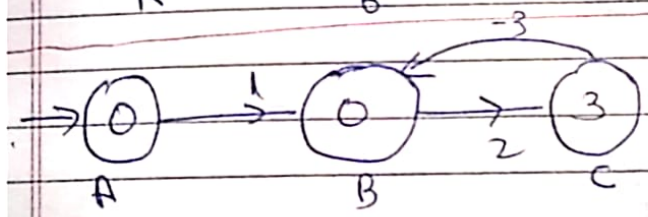
3 node ; 3-1 = 2



1 Relax



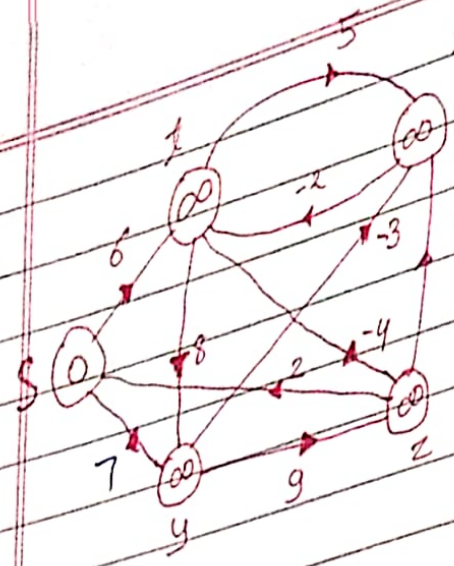
2 Relax

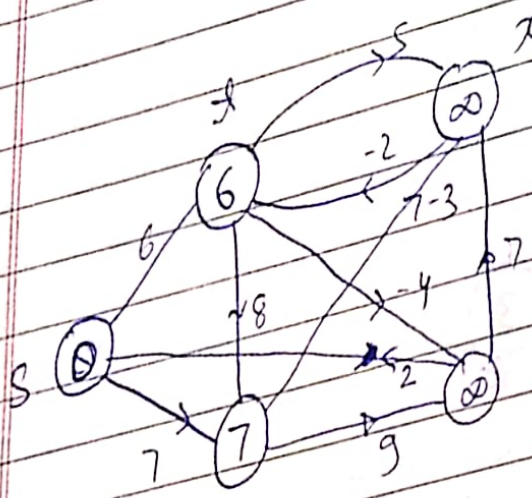


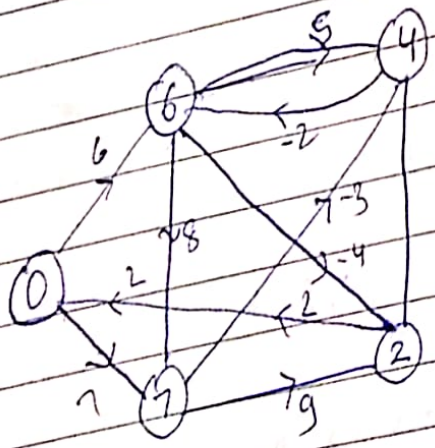
By going one more relax, it find the more shorter path. Hence there is cycle negative

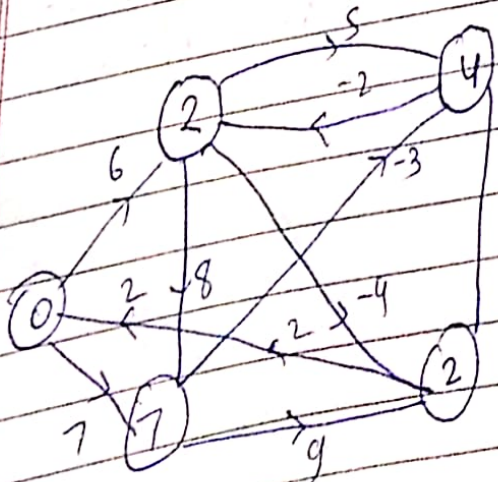
Time Complexity : O(VE)

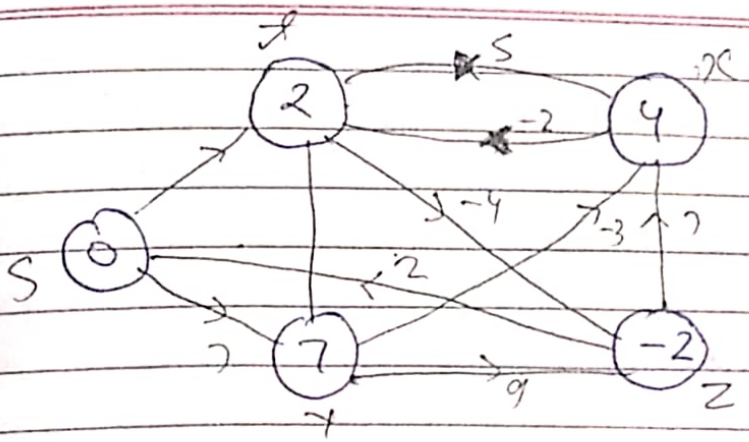
At every relax, all edges will be included. If there V relax, there will be O(VE)











Algorithm:

```
BELLMAN-FORD (G, W, S)
{
Initialize Single source (G, S)        // O(V)
For i=1 to |G.V|-1                     (V-1)
    for each edge (u,v) ∈ G.E           (O(E)
        RELAX (u, v, w).

For each edge (u,v) ∈ G.E
        if (v.d > u.d + w(u,v))
            return FALSE.
return TRUE
}
```

Hence Time complexity : $O(V-1) \times E$

$= O(VE)$

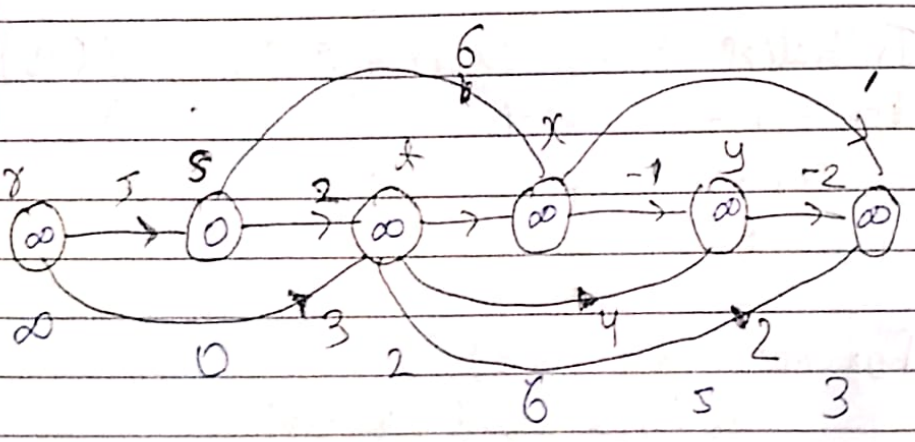
DAG : Directed Acyclic Graph :

If a graph doesn't have any cycle in it, then we can apply DAG.

DAG - Shortest - paths (G, w, s)

{

1. Topological sort the vertices of 'G' — $O(V+E)$
2. Initialize - Single Source (G, S) — $O(V)$
3. for each vertex u, taken in topologically sorted order
4. for each vertex $v \in G.adj[u]$
5. Relax (u, v, w)



* We will traverse all the node 'n' times.

Time complexity : $O(V+E)$

Dynamic programming:

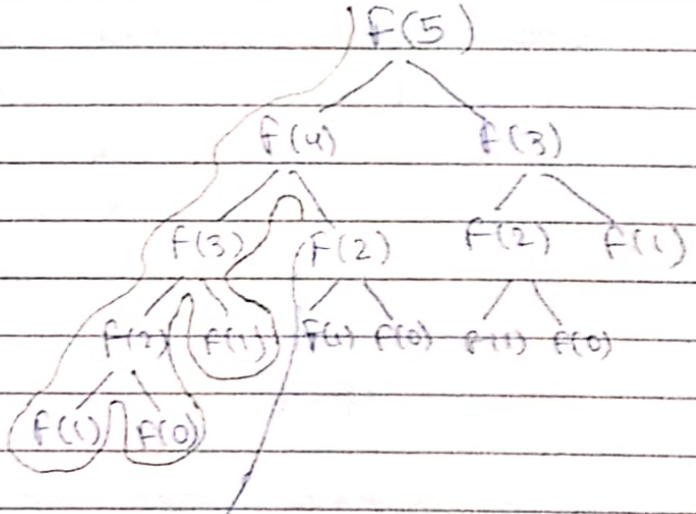
Whenever we do recursive call or function call before that call we will check in our table weather that data is present or not. If it is present it will save our lot of time. It is called dynamic programming when already computed result is used in f^n call. We use previous result to obtain new result if the same f^n call with same parameter will be called.

Fibonacci Series

$f(n) = f(n-1) + f(n-2)$, otherwise
$= 1$; $n == 1$
$= 0$; $n == 0$

```
f(n)
{
if (n==0)
    return 0;
if (n==1)
    return 1;

return ( f(n-1) + f(n-2) )
}
```



f(2) has already called, so wherever we call a f^n, we look at table weather its result is present or not. It will help us to decrease the no of call f^n.

Total f^n call = 2^n

$O(2^n)$

At this place we can write something like this:

Check the table for T(n)
if T(n) is absent
    f(n)
if T(n) is present
    f(n) = T(n)

2	3	4
1	2	3

Table

Matrix Multiplication

$A(3\times2)$ $B(2\times3)$

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix} \quad \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{bmatrix}$$

Total no entry $\begin{bmatrix} (a_{11}b_{11}+a_{12}b_{21}) & & \\ & & \\ & & \end{bmatrix}_{3\times3} = 9$

Total no multiplication we have to do

for 1 entry = 2 multiplication needed

for 9 entry = 2×9

$= 18$ multiplication needed

$$\begin{bmatrix} \quad \end{bmatrix}_{P\times R} \quad \begin{bmatrix} \quad \end{bmatrix}_{R\times Q}$$

So in general ~~case~~ we need = $[P\times R\times Q]$

Generally what we are learning here how we can decrease the no of operation so that complexity should decrease

$((A_{2\times1} \times B_{1\times2}) C_{2\times4})$

$2\times2\times1 = 4$

$2\times4\times2 = 16$

Total = 20

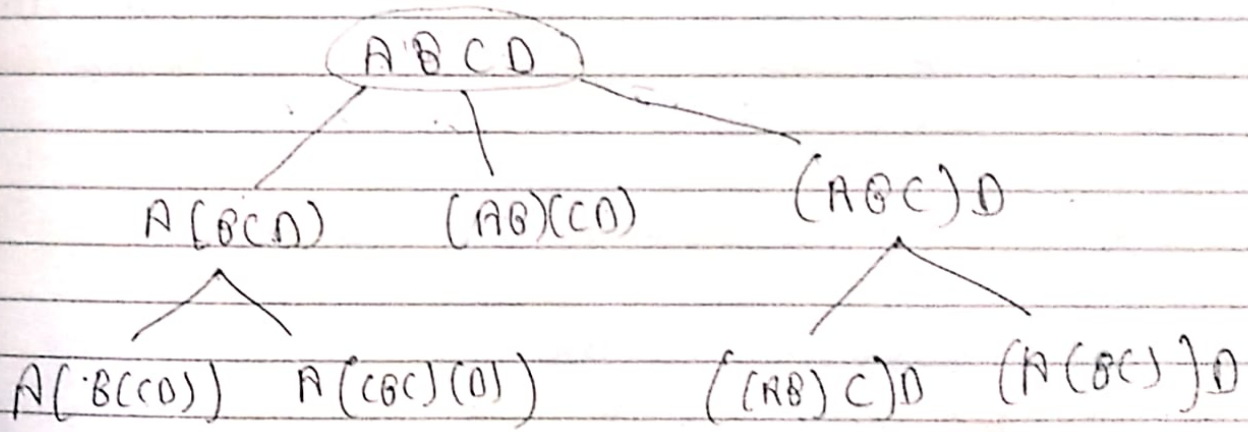
$(A_{2\times1} \times (B_{1\times2} \times C_{2\times4}))$

$1\times4\times2 = 8$

$2\times4\times1 = 8$

16

Hence in second method we have decrease the no of multiplication. It is the optimal way to use it a matrix or multiply a matrix.



Hence there is 5 ways to solve this.

One of this will be optimal soln.

2 = 1
3 = 2
4 = 5
5 = 14

$\frac{^{2n}C_n}{n+1}$, here n = (no' of element - 1)

$A_{1\times2}\ B_{2\times1}\ C_{1\times4}\ D_{4\times1}$

we have to solve it for every one, then find the optimal one.

Total 5 ways

(AB)(CD) = 7 , it is the optimal one

Optimal Substructure & Recursive Solution:

Here we use dynamic programming to know what is the optimal soln.

let us suppose that there are n-matrices

$A_1\ A_2\ A_3 \ldots A_n$

$p_0\times p_1 \quad p_1\times p_2 \quad p_2\times p_3 \ldots p_{n-1}\times p_n$

$A_i \ (p_{i-1} \times p_i)$

Now the parenthesis can come at any where b/w A_1 to A_n, parenthesis = k

$i \leq k < j$

They are having matrix with row & column

$$A_i | A_{i+1} | \cdots | A_j = (A_i A_{i+1} \cdots A_k)(A_{k+1} \cdots A_j)$$

$(p_{i-1} \times p_k)$ $(p_k \times p_j)$

$+ p_{i-1} p_k p_j$

Because when we multiply a matrix we always get the row of 1^{st} matrix & column of the last matrix.

This will be added as no of multiplication required.

Recursive equation:

$$m[i,j] = \begin{cases} 0 & ; i = j \\ \min \{ m[i,k] + m[k+1, j] + p_{i-1} p_k p_j \} & \end{cases}$$

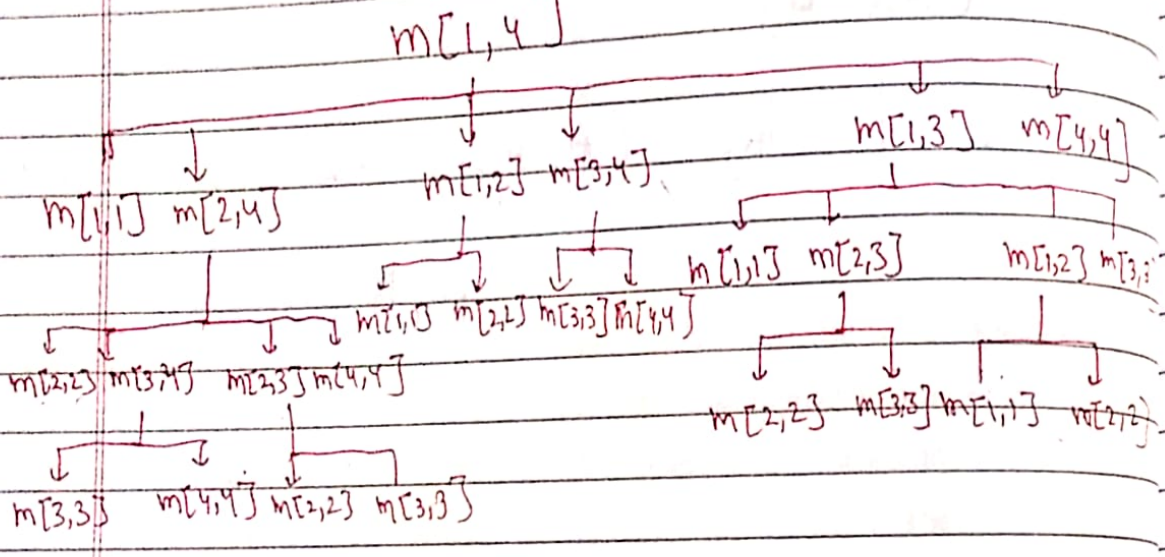
$i \leq k < j$

$A_i \ A_{i+1} \ A_{i+2} \cdots A_j$

$k \sim i, i+1, i+2 \cdots (j-1)$

Recursion Tree

$A_1 | A_2 | A_3 | A_4$



* We store the data of previous recursion so that we can further use it.
* By doing this we can decrease the complexity from exponential to $O(n^3)$

If we simple do this it is going to take $O(2^n)$, not exactly but somewhere closer to it.

	A(1,4)	A(1---11)
(1,1) (2,2) (3,3) (4,4)	- 4 (1)	n (1)
(1,2) (2,3) (3,4)	- 3 (2)	(n-1) (2)
(1,3) (2,4)	- 2 (3)	(n-2) (3)
(1,4)	- 1 (4)	(n) (n)
	4+3+2+1 = 10	$\frac{n(n+1)}{2} \approx O(n^2)$

NP = Non-deterministic polynomial

These are total $O(n^2)$ entry's to fill one entry in worst case take $O(n)$ time, the last entry (1,4) because there will be (n-1) split, Hence

$$O(n^2) \times (O(n)) = O(n^3)$$

Bottom-up method: (Iterative method)

$(1\,1)_0$	$(2\,2)_0$	$(3\,3)_0$	$(4\,4)_0$
$(1\,2)_2$	$(2\,3)_8$	$(3\,4)_4$	
$(1\,3)_4$	$(2\,4)_6$		
$(1\,4)_7$			

→ Total no of entry

worst case to fill each

$O(n^2) \times (O(n))$
$= O(n^3)$

A_1 A_2 A_3 A_4
1×2 2×1 1×4 4×1
p_0 p_1 p_1 p_2 p_2 p_3 p_3 p_4

$p_0=1, p_1=2, p_2=1, p_3=4, p_4=1$

$$(1,2) = \left\{ \min \left\{ \underset{0}{(1,1)} + \underset{0}{(2,2)} + p_0 \times p_1 \times p_2 = 2 \right\} \right.$$

$$(2,3) = \min \left\{ \underset{0}{(2,2)} + \underset{0}{(3,3)} + p_1 \times p_2 \times p_3 = 8 \right\}$$

$$(3,4) = \min \left\{ (3,3) + (4,4) + p_2 \times p_3 \times p_4 = 4 \right\}$$

$$(1,3) = \min \left\{ \begin{array}{l} (1,1) + (2,3) + p_0 p_1 p_3 = 8 \\ (1,2) + (3,3) + p_0 p_2 p_3 = 4 \end{array} \right\}$$

$$\min \left\{ \begin{array}{l} (2,2) + (3,4) + p_1 p_2 p_4 = 6 \\ (4,4) + p_1 p_3 p_3 = 16 \end{array} \right\}$$

no of terms

1 2 3

Time complexity = $(n-1) + (n-2) + (n-3) + \cdots$

Total time to split from last

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

$= n[1+2+3+\cdots n] - 1 - 2 - 3 \cdots$

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

$= O(n^3)$, Space complexity = $O(n^2)$

```
MATRIX-CHAIN(p)
{
1  n = p.length - 1
2  let m[1...n, 1...n] & s[1...n-1, 2...n] be new tables
3  for i=1 to n
4      m[i,i] = 0
5  for l=2 to n  // l is the chain length
6      for i=1 to n-l+1
7          j = i+l-1
8          m[i,j] = ∞
9          for k=i to j-1
10             q = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j
11             if q < m[i,j]
12                 m[i,j] = q
13                 s[i,j] = k
14 return m & s
```

Iterative matrix

Top-down matrix Multiplication

(Recursive method)

```
MEMOIZED-MATRIX-CHAIN(P)
{
 n = p.length - 1
 let m[1...n, 1...n] be a new table
 for i = 1 to n
     for j = 1 to n
         m[i,j] = ∞
 return LOOKUP-CHAIN(m, p, 1, n)


LOOKUP-Chain(m, p, i, j)
{
 if m[i,j] < ∞
     return m[i,i]
 if i == j
     m[i,i] = 0
 else for k = i to j-1
     q = LOOKUP-Chain(m, p, i, k)
         + lookup-Chain(m, p, k+1, j) + p_{i-1} p_k p_j

 if q < m[i,j]
         m[i,j] = q
 return m[i,j]
```

Time complexity = $O(n^3)$

Space complexity = $O(n^2) + O(n)$

($O(n^2)$: Matrix; $O(n)$: Stack)

$= O(n^2)$

Q1 M_1 M_2 M_3 M_4

(10×100) (100×20) (20×5) (5×80)

$(M_1M_2)(M_3M_4) = 20000 + 8000 + 16000$

$((M_1M_2)M_3)M_4 = 20000 + 1000 + 4000$

$(M_1(M_2M_3))M_4 = 10000 + 5000 + 4000 = 19000$

$M_1((M_2M_3)M_4) = 10000 + 4000 + 80000$

$M_1(M_2(M_3M_4)) = 8000 + 16000 + 80000$

Ans = 19,000

Longest Common Subsequence:

It is the sequence of a string in which character are present in subsequence in increasing order such that some of the char can be missed, but the subsequence should be in increasing order.

R A V I N D R A
1 2 3 4 5 6 7 8

{1, 2, 3, 4} · RAVI ✓

{1, 3, 5, 6} = RVND ✓

{3, 6, 7, 8} - VDRA ✓

{3, 6, 5, 7} VDNA ✗
→ Due to this.

{} = this is also subsequence

If a string of length 'm', total number of possible sequence are
$= 2^m$.

It is used in DNA matching.
We try to find out the longest subsequence.

RAVINDRA $= 2^m$

AJAY $= n$

{AA}
{A}
{Ø}
} 3 possible, {AA} would be selected.

$O(2^m)$ 1. Find all the subsequences of 'A'.

$O(n.2^m)$ 2. Find for each subsequence whether it is a subsequence in 'B'

$O(2^m)$ 3. Find the longest among common subsequence.

Time complexity: $O(n.2^m)$

To decrease the time complexity, we are going to use dynamic programming to find optimal substructure:

X: $x_1\ x_2\ - - - - x_n$

Y: $y_1\ y_2\ - - - - - y_m$

If $(x_n = y_m)$, then we will do comparison from $(x_1 - - x_{n-1})$ to $(y_1 - - y_{m-1})$

If $(x_n != y_m)$, then we will do comparison from $(x_1 - - x_{n-1})$ to $(y_1 - - y_m)$ or $(x_1 - - x_n)$ to $(y_1 - - y_{m-1})$

Because if we have done a comparison one time of a char with other, there is no mean to check it again with the same. We started from the last.

$x_1, x_2 \; - - - \; x_i$

$y_1, y_2 \; - - - \; y_j$

$$C[i,j] = \begin{cases} 0, & i=0 \text{ or } j=0 \\ 1 + C[i-1, j-1]; & i,j>0 \;\&\; x_i = y_j \\ \max\{C[i-1,j], C[i,j-1]\}; & i,j>0 \;\&\; x_i \neq y_j \end{cases}$$

$X = \{A\ A\ AA\}$

$Y = \{A\ A\ AA\}$

1+ Case

C(4,4)

1 + C(3,3)

1 + C(2,2)

1 + C(1,1)

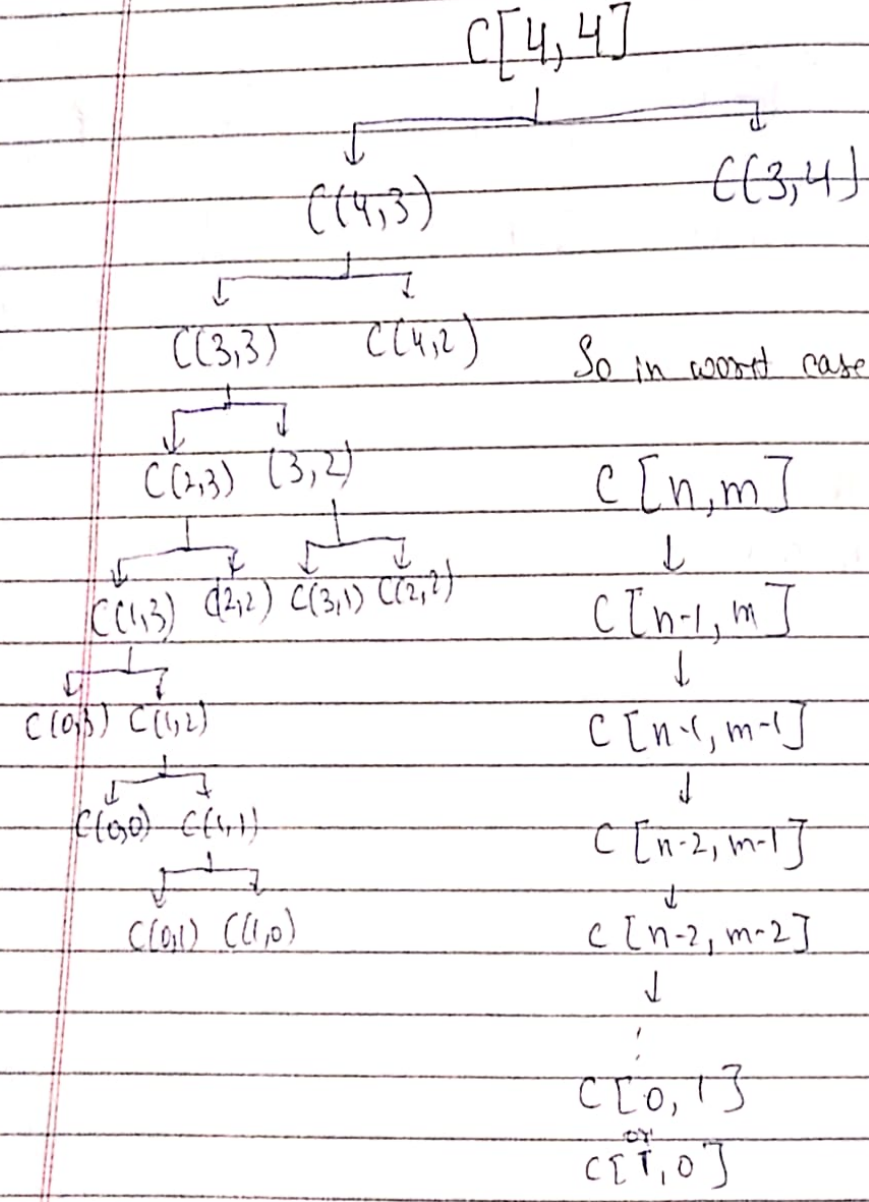
1 + C(0,0)

Hence in 4 case we find it

$X = \{A, A, A, A\}$

$Y = \{B, B, B, B\}$

This is worst case



So in worst case, the height of tree

C[n, m]

↓

C[n-1, m]

↓

C[n-1, m-1]

↓

C[n-2, m-1]

↓

C[n-2, m-2]

↓

⋮

C[0, 1]

or

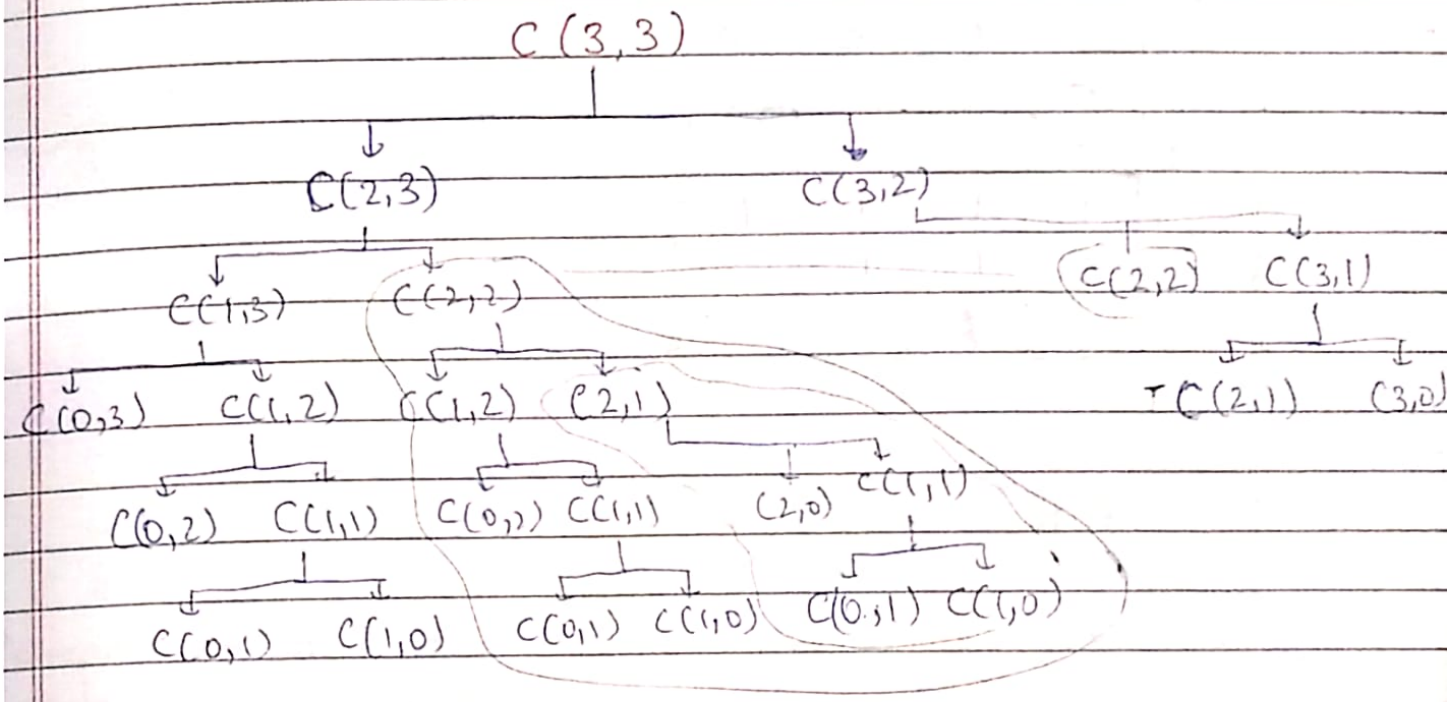
C[1, 0]

So height of tree goes to $O(n+m)$

Total no of node = $O(2^{n+m})$ in worst case time complexity.

But there are some f^n which are calling again & again, so we have to identify those f^n & maintain there table, so that we don't have to go to recursion again.

Now, taking an example of two string of length 3 & 3.



So we have to find all the unique fn which will be repeated.

(3,3) (3 2) (3 1)
(2 3) (2 2) (2 1)
(1 3) (1 2) (1 1)
(1 0) (2 0) (3 0)
(0 1) (0 2) (0 3) (0 0)

For 3, 3
it will be (3+1)(3+1)
= 4

	0	1	2	3
0	00	01	02	03
1	10	11	12	13
2	20	21	22	23
3	30	31	32	33

, first we will solve 00, 01, ... 03
Before solving any node we will solve
They all should be solved

O(mn), total blocks & it take just constant time to fill them, Time complexity = O(mn)
Space " = O(mn)

* X (A A B)
Y (A C A)

	Y	A	C	A
		0	1	2	3
X	0	0	0	0	0
A	1	0	1	1	1
A	2	0	1	1	2
B	3	0	0	1	2

longest subsequence 2

(A A) answer

(1) Whenever there is a match we add 1 to block, a[1,1] = 1 + a[0,0] whenever there

(2) If there is no match we take maximum left & upper block. For a[1 2]

* At last block, that will be answer.
* we started traversing from there, how w
when there is a match, we started
right to left.

X = { A, B, C, B, D, A, B }
Y = { B, D, C, A, B, A }

		Y	B	D	C	A	B	A
	X	0	0	0	0	0	0	0
A		0	0	0	0	1	1	1
B		0	1	1	1	1	2	2
C		0	1	1	2	2	2	2
B		0	1	1	2	2	3	3
D		0	1	2	2	2	3	3
A		0	1	2	2	3	3	4
B		0	1	2	2	3	4	4

→ BCBA

its diagonal
re IR match.

BDAB, longest BDAB
or
BCAB

3 string are possible.

ence of
] = max [a[i-1], a[i]

we get that,
letting from

```
LCS (X, Y)
{
m = X.length
n = Y.length
let C[0...m, 0...n] be a new table
for i = 1 to m
    C[i,0] = 0
for j = 0 to n
    C[0,j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            C[i,j] = C[i-1, j-1] + 1
        else
            C[i,j] = max( C[i-1, j] or C[i, j-1] )

return C
}
```

Time complexity : O(mn)
Space complexity : O(mn)

14-GATE

A = q p q r r

B = p q p r q r p

X + 10y , x = length of highest subsequence,
Y = total no' of highest subsequence.

		A	1	2	3	4	5
		0	q	p	q	r	r
B	0	0	0	0	0	0	0
1	p	0	0	1	1	1	1
2	q	0	1	1	2	2	2
3	p	0	1	2	2	2	2
4	r	0	1	2	2	3	3
5	q	0	1	2	3	3	3
6	r	0	1	2	3	4	4
7	p	0	1	2	3	4	4

p q r r
q p q r
q p q r
q p r r

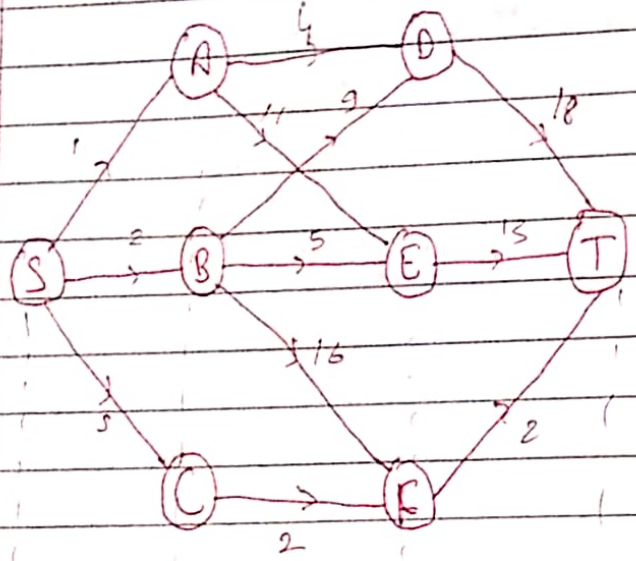
x = 4

y = 3

x = 10 4 + 10 x 3 = 34

Multistage graph:

These are the graph in which cycle is not present, & graph can be divided into various stage. An edge can go from itself to its child node & to it's sibling's child node, not to his sibling. Then that graph is called multistage graph.



I stage II nd stage III IV

If we apply Dijkstra here, the it find the shortest path to the all other given node. Time complexity will also high ($E \log v$)

* By applying greedy, we get answer wrong.

$S \xrightarrow{1} A \xrightarrow{4} D \xrightarrow{18} 23$

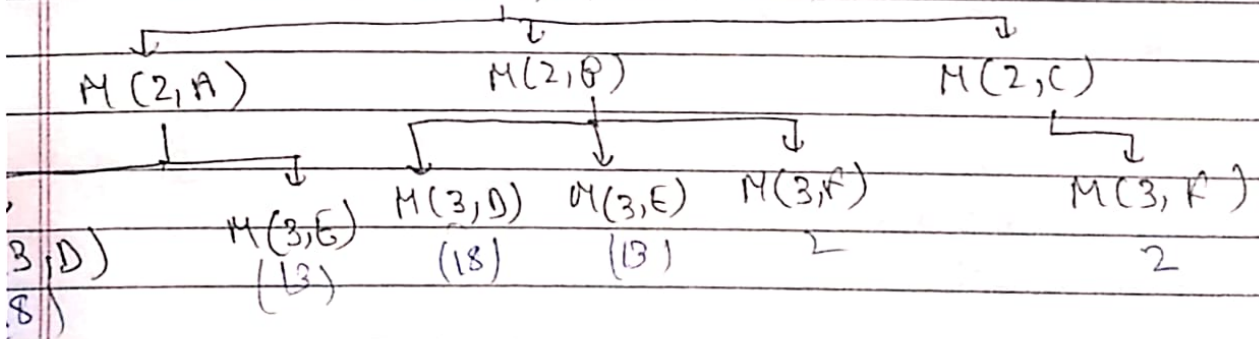
M(3
(18

Now we will try to solve it with dynamic programming
- If we get any repeated thing, we will save it for later.
- We can decrease the time complexity also.

M(I,S) = Minimum in I^st stage to S.
It will give us result of shortest path from source to destination

$$M(1,S) = \min \begin{cases} S \to A + M(2,A) \\ S \to B + M(2,B) \\ S \to C + M(2,C) \end{cases}$$

(stage, node)



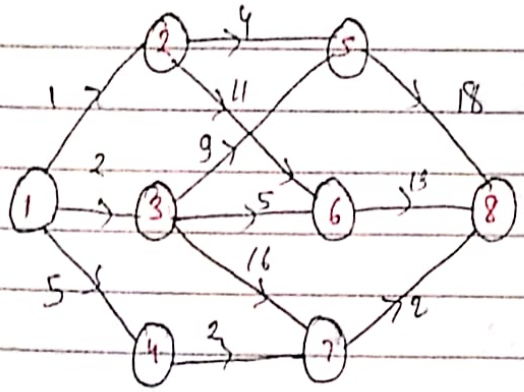
These are some f^n call which are repeating. Hence we store that f^n result.

O(K), Total stage & O(V) = Total vertex.

- We will create an array S started from the destination node to source node.
- We make it zero, then go to its ~~predecessor~~ ~~successor~~ predecessor.
- We apply insertion sort type algo there, we will apply it on its successor.
- We numbered all the vertex from 1 to n.
- The minimum cost will be filled in array.

T

1	2	3	4	5	6	7	8
9	22	18	4	18	13	2	0



$$T[i] = \min_{\text{for all } j \in (i+1) \text{ to } n} \{ Cost(i,j) + T[j] \}$$

if there is no direct edge we will take it infinite.

$$T[7] = \{ C(7,8) + T[8] \}$$

$$= 2 + 0 = 2$$

$$T[6] = \min \begin{cases} \overset{\infty}{Cost(6,7)} + \overset{2}{T[7]} = \infty \\ \underset{13}{Cost(6,8)} + \underset{0}{T[8]} = 13 \end{cases}$$

$$T[5] = \min \begin{cases} \overset{\infty}{Cost(5,6)} + \overset{13}{T[6]} = \infty \\ \overset{\infty}{Cost(5,7)} + \overset{2}{T[7]} = \infty \\ \underset{18}{Cost(5,8)} + \underset{0}{T[8]} = 18 \end{cases}$$

$$T[4] = \min \begin{cases} \overset{\infty}{Cost(4,5)} + \overset{18}{T[5]} = \infty \\ \overset{\infty}{Cost(4,6)} + \overset{\infty}{T[6]} = \infty \\ \overset{2}{Cost(4,7)} + \overset{2}{T[7]} = 4 \\ \underset{\infty}{Cost(4,8)} + \underset{0}{T[8]} = \infty \end{cases}$$

Hence Time complexity is same as insertion sort

$$O(n^2) = O(V^2) = O(6)$$

$$1 + 2 + \cdots n-1 = \frac{n(n-1)}{2} = O(n^2) = O(V^2) = O(6)$$

Space complexity = $O(V)$

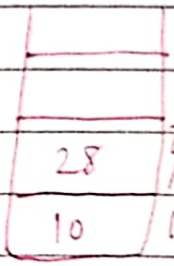
Knapsack Problem:

- Fraction (We can apply greedy next)
- 0/1 (But can't apply greedy, because it will give different result)

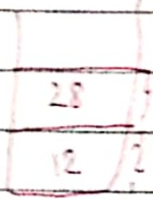
We are filling object in bag. We can't do it fraction. Then see what will happen when we apply greedy method.

Capacity = 6

Objects	1	2	3
Weight	1	2	4
Profit	10	12	28
P/w	10	6	7



38 (By greedy)



40, without greedy.

Hence we won't apply greedy in fraction those object where fractioning is not allowed.

(1)

(2)

(3)

$KS(i, w)$ = i = no' of element or object
w = capacity of the bag
p_i = profit associated with object
w_i = weight of the object.

$$KS(i,w) = \begin{cases} \max\left(p_i + KS(i-1, w-w_i),\ KS(i-1, w) \right) \\ 0 \quad ; \ i = 0 \text{ or } w = 0 \\ KS(i-1, w) \quad ; \ w_i > w \end{cases}$$

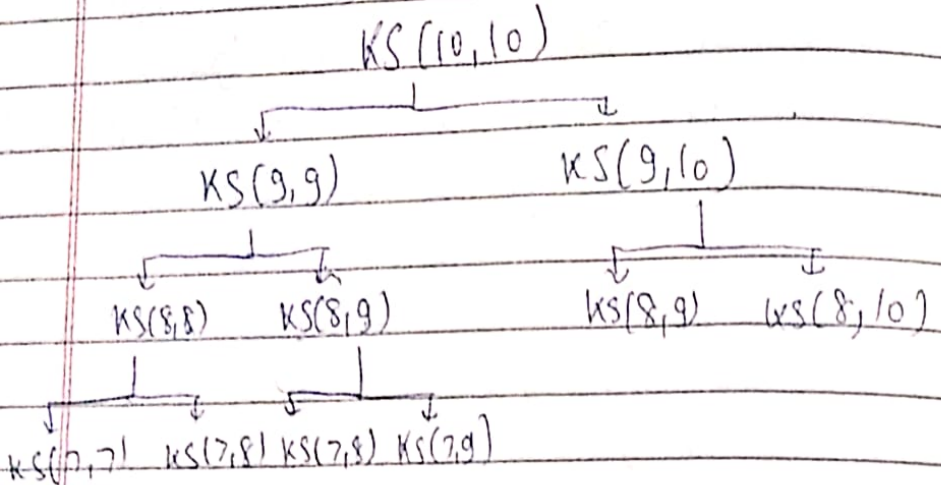
we take maximum of it if we are getting profit after inserting that element, or after removing that object if we get the max profit. We will choose max of these 2

if either $i=0$, no object left or no capacity of bag is left, we stop it.

if the weight of object is greater than capacity of bag left, we simply eliminate that object & move to next one.

10 objects → Capacity of bag. Every object weight = 1

KS(10,10)



KS(0,0)

Bottom-up approach

n = no of object

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

But here the problem are repeating

= 10 x 10 = 100 no of unique call, other are repeating

= $O(n \times w)$ ← weight of bag

By

$$KS(i,w) = \begin{cases} \max(b_i + KS(i-1, w-w_i), KS(i-1,w)) \\ 0, \quad i=0 \text{ or } w=0 \\ KS(i-1,w), \quad w_i > w \end{cases}$$

C = 6

	1	2	3
W	1	2	4
b	10	12	28

		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
10	1	0	10	10	10	10	10	10
12	2	0	10	12	22	22	22	22
28	3	0	10	12	22	28	38	40

← Total profit

$$KS(1,1) = \begin{cases} 10 + KS(0,0) \\ KS(0,1) \end{cases}$$

$$KS(1,2) = \begin{cases} 10 + KS(0,1) \\ KS(0,2) \end{cases}$$

$$KS(2,1) = \{ KS(1,1) \}$$

taking 2 object, can 1 space can be filled

$$KS(2,2) = \begin{cases} 12 + KS(1,0) = 12 \\ KS(1,2) \end{cases}$$

$$KS(2,3) = \begin{cases} 12 + KS(1,1) = 22 \\ \end{cases}$$

So here time complexity = $O(nw)$

but when w becomes $= 2^n$, then we won't use this method because it become large $(n \cdot 2^n)$. On that case i use brute force $O(2^n)$

Time comp^{lx} $\min\{O(nw), O(2^n)\}$

If $w < 2^n$, use $O(nw)$

else $w = 2^n$, use $O(2^n)$

Space complexity. $O(nw)$

Algorithm:

input : $\{w_1, w_2, w_3 \dots w_n\}, c, \{p_1, p_2, \dots p_n\}$

output : $T[n, c]$

```
for i=0 to c do
    T[0,i]=0
for i=1 to n
{   T[i,0]=0
    for (j=1 to c) do
        if (wi<=j) & T[i-1,j-wi]+pi > T[i-1,j]
        then T[i,j] = T[i-1,j-wi]+pi
        else
            T[i,j] = T[i-1,j]
}
```

Subset Sum:

Given a set of 'n' no, if there is set of no whose sum is equal to given no

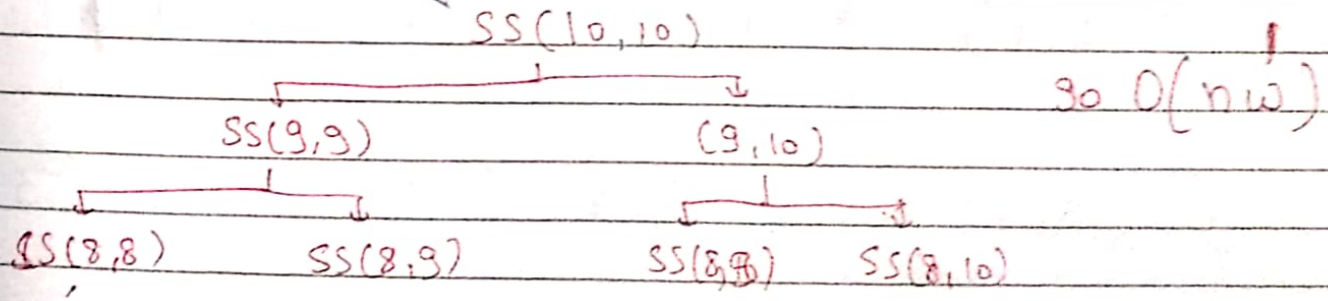
$\{a_1, a_2 \ldots a_n\}$ — "Set of no" — "W"

$\{6, 2, 3, 1\}$

Brute force method; 2^n subset construct then compare, complexity will be high.

$$SS(i, S) = \begin{cases} SS(i-1, S) & ; S < a_i \\ SS(i-1, S-a_i) \vee SS(i-1, S) & ; S \geq a_i \\ \text{// Here we find no' is present or not, it can be present in any sequence, that why we take } \vee. \\ \text{True} & ; S = 0 \\ \text{False} & ; i = 0 ; S \neq 0 \end{cases}$$

$SS(i, S)$: i → no of element; S → Sum required



So $O(nw)$

Special point:

* We are stasting from the bottom, weather to take the number or not.
* That base condition will be present with every leaf node.
* In this way we move upwards, towards the root.

* ~~Then~~ Weather to include the number or not.
* It will start from top, & go till bottom.
* In this way we get all the cases.

$\{6, 3, 2, 1\}$
$w = 5$

Nos of Object taken order		0	1	2	3	4	5
6	0	T	F	F	F	F	F
3	1	T	F	F	F	F	F
2	2	T	F	F	T	F	F
1	3	T	F	T	T	F	T
	4	T	T	T	T	T	T

$SS(1,1) = \{$ as Sum < a; so $SS(0,1)$ $1 < 6$ $\}$

Total object, total Sum

$SS(2,1) = \{ 1 < 3 \text{ , so } SS(1,1) \}$

$SS(4,3) = \{ Sum > 1, SS(3,2) \cup SS(3,3) \}$

no of object → Total sum required

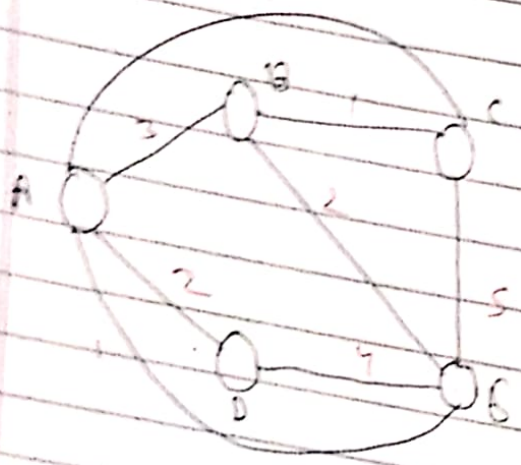
Time complexity = $O(nw)$

If $w = 2^n$, in that case we don't use this method, we use brute force.

Time complexity = $O(2^n)$

Traveling Salesman Problem

There is a ~~beast~~ salesman who want to travel every village & back to its own place. The shortest path he will take to traverse all the village & back to its own place.



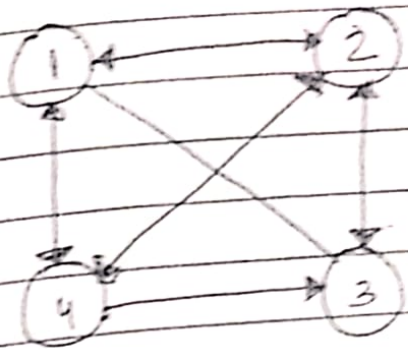
Hamiltonian cycle: Cost should be minimum

A B C D E A

$(n-1)!$

$O(n!)$ $\quad n! \geq 2^n$

So it is worst case.



	1	2	3	4
1	0	1	2	3
2	1	0	4	2
3	1	2	0	5
4	3	4	1	0

By default cost = 0

$$T(1,\{2,3,4\}) = \min \begin{cases} (1,2) + T(2,\{3,4\}) \\ (1,3) + T(3,\{2,4\}) \\ (1,4) + T(4,\{2,3\}) \end{cases}$$

$= 5$

$$T(2,\{3,4\}) = \min \begin{cases} (2,3) + T(3,\{4\}) = 12 \\ (2,4) + T(4,\{3\}) = 4 \end{cases}$$

$$T(3,\{2,4\}) = \min \begin{cases} (3,2) + T(2,\{4\}) = 7 \\ (3,4) + T(4,\{2\}) = 10 \end{cases}$$

$$T(4,\{2,3\}) = \min \begin{cases} (4,2) + T(2,\{3\}) = 9 \\ (4,3) + T(3,\{2\}) = 4 \end{cases}$$

$T(3,\{4\}) = (3,4) + T(4,\emptyset) = 8$

$T(4,\{3\}) = (4,3) + T(3,\emptyset) = 2$

$T(2,\{4\}) = (2,4) + T(4,\emptyset) = 5$

$(4,2) + T(2,\emptyset) = 5$

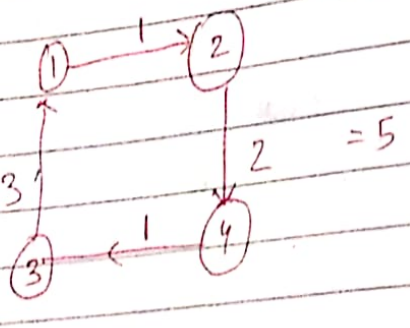
$T(2,3) = (2,3) + T(3,\emptyset) = 5$ (4, 1)

$T(3,2) = (3,2) + T(2,\emptyset) = 3$ (2, 1)

$T(2,\emptyset) = (2,1) = 1$

$T(3,\emptyset) = (3,1) = 1$

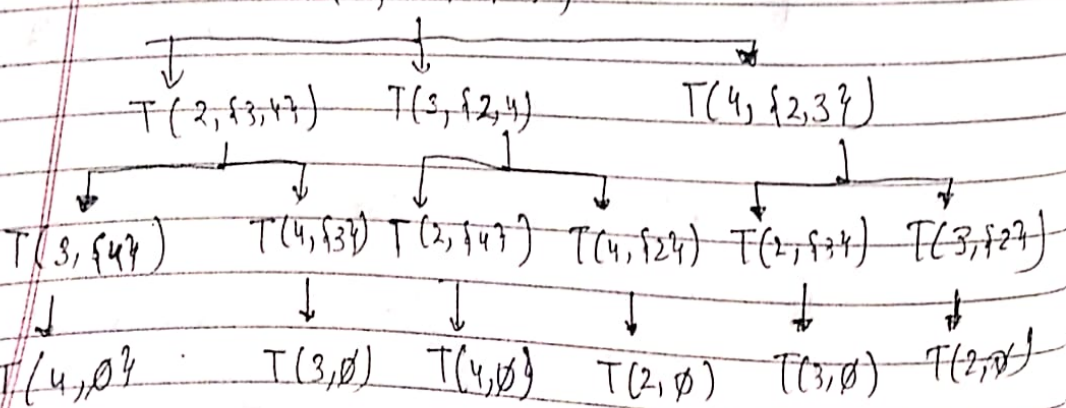
$T(4,\emptyset) = (4,1) = 3$



Recursive equation for this

$$T(i,S) = \min_{j \in S} \left(C(i,j) + T(j, S-\{j\}) \right) ; S \neq \emptyset$$

$$= (i,1) \quad ; \quad S = \emptyset$$



Only bottom are repeating, So they does not effect them by too much

Node from where we have to traverse.

Set of all remaining node

discrete

Date
Page

initial $T(i, S)$ n = Total no of vertex or no of

no of value

$(n-1)$ [This will be present at every level]

Generally we are finding total no of recursive call.

Now at first level, they are $(n-2)$,
IInd level, they are $(n-3)$
:
0

Now at {2, 3 & 4}, we make its set at every level as S. Somewhere we select all three, somewhere 2, somewhere 1. They can be selected in different way. Start from I^st level

$$(n-1) + (n-1)^{n-2}C_{n-3} + (n-1)^{n-2}C_{n-4} + \cdots$$

$$(n-1)\left[1 + {}^{n-2}C_{n-3} + {}^{n-2}C_{n-4} + \cdots\right]$$

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

$O(n \cdot 2^n)$ = Total no of recursive call

n subset will get $(n-1)$ no, it has to choose minimum of them, so it will take $O(n)$ time.

Time complexity - $O(n \cdot 2^n) \cdot O(n)$

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

Space complexity = $O(n \cdot 2^n)$ (Table which we have to create)

$O(n)$ = Entry to fill on each by doing $(n-1)$ comparis

All pair Shortest path: FLOYD Warshall

We have to find shortest path to every node from a given node. In this way we will find shortest path for every node.

$V = \{1, 2, 3 \ldots n\}$

$i, j \in V$

$d_{ij}^{K} = ("b"$

distance of i to j including K vertices $\{1, 2, \ldots K\}$

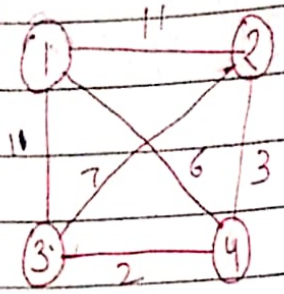
→ $d_{ij}^{(K-1)}$ // does not include K^{th} vertex

→ $d_{iK}^{(K-1)} + d_{Kj}^{(K-1)}$ // distance by going through K

$(K-1)$, K is not b/w the path.

i 1 2 3 K

K is not in b/w



D^0 = Direct distance of a node to other node.

D^1 = Distance of any node to others but pass through 1

D^2 = Distance of any node to other including 2 or 1 or both

∴ we always checked from the previous table

D^0

	1	2	3	4
1	0	11	1	6
2	11	0	7	3
3	1	7	0	2
4	6	3	2	0

D_1

	1	2	3	4
1	0	11	1	6
2	11	0	7	3
3	1	7	0	2
4	6	3	2	0

$d_{22} \rightarrow 2 \rightarrow 1, 1 \rightarrow 2 = 22$ (11, 11)

it is given from previous value

$d_{23} \rightarrow 2 \rightarrow 1, 1 \rightarrow 3 = 12$ (11, 1)

$7 < 12$

D_2

	1	2	3	4
1	0	11	1	6
2	11	0	7	3
3	1	7	0	2
4	6	3	2	0

$d_{13} \rightarrow 1 \rightarrow 2 + 2 \rightarrow 3 = 18$ (11, 7)

$d_{14} \rightarrow 1 \rightarrow 2 + 2 \rightarrow 4 = 14$ (11, 3)

(1) First ~~allot the~~ fill the diagonal with zero
(2) Fill all the element of a node directly same as previous one whom we are traversing. fill directly.

D_3 =

	1	2	3	4
1	0	8	1	3
2	8	0	7	3
3	1	7	0	2
4	3	3	2	0

$d_{12} \rightarrow (1 \rightarrow 3), (3 \rightarrow 2)$

D_4 =

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

$d_{23} \rightarrow (2 \rightarrow 4), (4 \rightarrow 3)$
$3 \quad 2 = 5$

Time complexity: $O(n^2) \times n$ → total matrix we have to calculate (n° of entries in one matrix)
$= \Theta(n^3)$

Space complexity: $O(n^2)$, at a time we need only 2 matrix. One present & one previous

```
FLOYD-WARSHALL (W)
{
  n = W.rows
  D^0 = W
  For k = 1 to n
  Let D^(k) = (d_ij^k) be new nxn matrix
    For i = 1 to n
      for j = 1 to n
        d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
  return D^(n)
```

NP-completness problem:

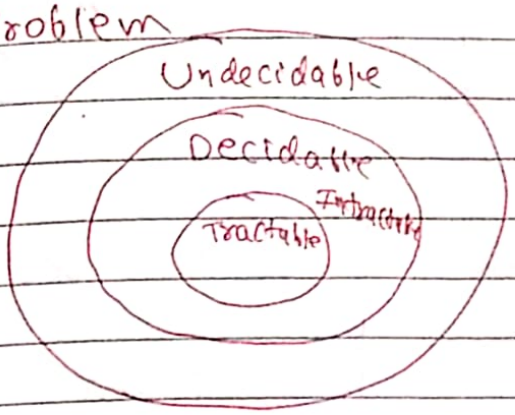
* There are some problem for which machine does not halt in polynomial time $O(n^k)$
* They take too much time to show output even that output is possible take days to solve
* If there is a problem whose solution is not find yet in polynomial time & assuming that no-other can solve it
* But in future it may be possible to solve the problem.

Problem

- Decidable problem (algo exist)
  - Tractable
    * If there exist at least one polynomial bound algorithm.
    - $O(n^k)$
    - $O(n)$ & $O(n^2)$
    - $O(n^{1000})$
  - Intractable
    * If a problem is not tractable then it is intractable
    - $O(n^{\log n})$ $O(c^n)$
- Undecidable problem (no algo exist)
  - Like halting problem of TMS

// $O(n^{1000})$ can be reduce in future to some extent & we get better result

Problem



By just looking at the problem we can't decide that it will be tractable or Intractable.
So we have to follow some rule to know that problem is having polynomial time sol^n or not

Shortest path = $O(E \log v)$
Longest path = $O(n!)$
We know the time complexity of Shortest path, but when we look at Longest path we can think that it can also be solvable in polynomial time but that is not true.

Euler Tour - $O(E)$
Hamiltonian cycle - $O(n \cdot 2^n)$ (Traveling salesman problem)

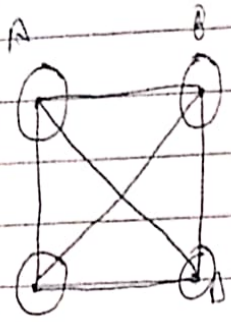
2CNF - 3CNF
↓
conjunctive normal form

$(x_1 \vee x_2) \wedge (x_3 \vee x_4)$ Polynomial time

$(x_1 \vee x_2 \vee x_3) \wedge (x_4 \vee x_5 \vee x_6)$ exponential time

MIN = Atmost

Optimisation & Decision Problem

Optimisation Problem	Decision problem
	Yes/No
TSP: A Graph 'G', Shortest path coresing all vertices exactly once. A-B-C-D-A	Is there any SP covering all the vertices of length atmost K. yes / NO
0/1 Knapsack :- Given C, p, W, find out the max profit.	Is there any solution whose profit is atleast "K"
LCS : A, B. Find LCS accde acdae	Is there any subsequence whose length is atleast K.

* First we decrease the level of problem in Decision problem.
* If we are not able to solve decision problem in polynomial time so we can't solve optimisation problem also in Polynomial time.

Optimisation → Decision problem.

TSP
10

At most 15
If it can be solve, the TSP also solve in polynomial time.

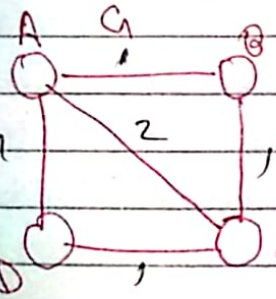
If Optimisation is easy ($O(n^k)$)
the DP is easy.

If Decision problem is hard
then Optimisation is also hard.

Verification Algorithm:

By providing the qvertion
By providing its solution also
By providing its Decision problem

Then we have to check the solution with the help of Decision problem, if we get the answer in polynomial time, it means the O.P is easy.



Decision problem
Is this graph hamiltonian

Solution
A → B → C → D - A

Now we have to verify weather D.P can be answered.

(1) All the vertex are present ✓
(2) The edges are present ✓

* If we have O.P ∆ corresponding to its D.P
* Now suppose O.P has easy solution in Polynomial time
* Now in polynomial time D.P has to verified that solution.
* i.e if OP is easy the DP is also easy

	D.P	Solution
Graph given.	TSP - Atmost 5	A → B → C → D → A (4)

Yes answer.

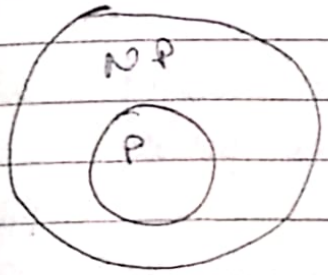
Decision problem

P: Set of all, polynomial time algorithm to solve them.

NP: Set of all decision problem for which there is a polynomial time verification.

P	NP
G, MST atmost 10? yes/no because prims, kruskal are there for Optimal sol^n, so DP can be easy	⟹ If we provide sol^n, we can definitely verified in polynomial time (Tractable)
Fractional, KS, profit is atleast 10? yes/no; Greedy algo	Similarly Here.

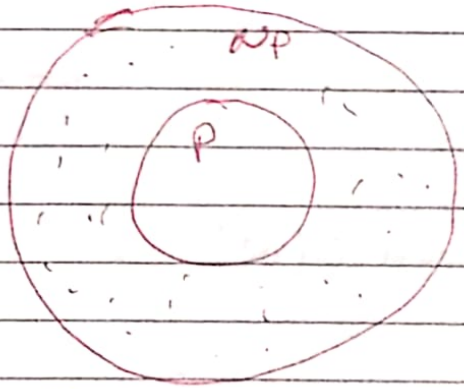
TSP does not work in polynomial time.	But providing soln we can verified it in polynomial time. (Intractable)



We are assuming it is true, but no one proved it yet.

If we able to proof that not any one NP problem can't be solve in Polynomial time.

If TSP can't be solve in Polynomial time, if we prove that, then we can say P ⊂ NP.



Problem in P, can be quickly solved or easily solved

NP → quickly verified or easily verified.

In NP, if we want to show that the every problem in NP can be solved in polynomial time, then we choose the hardest problem & solve it in polynomial time.

Here we use the concept of reducibility. We convert one problem into other problem, if we can solve that problem it will implies that we can solve the previous one in polynomial. It is not optimisation or decision problem.

A problem 'A' is said to be polynomial time reducible to a problem 'B' if

(i) Every instance 'α' of 'A' can be transformed to some instance 'β' of B in polynomial time.

(ii) Answer to 'α' is 'yes' if & only if answer to 'β' is 'yes'.

1-) A $\xrightarrow{\text{"p"}}$ B → It take $O(n^d)$

α β

$O(n^k)$

Then A can be solved in polynomial time, because $O(n^k)$ to transform & $O(n^d)$ to solve.

* But if $p = O(2^n)$, B take $O(n^d)$ then, we does n't say anything about A

No problem has been found that any algo does not have polynomial time.

If 'B' is easy, then A is also easy (According to)

If "B" is in 'P', then 'A' is also in 'P'

$A \xrightarrow[O(n^x)]{} B$

↓

of P"

If A is not in 'P', then 'B' is not in 'P'

↓

Proving this is very hard, we are just assuming.

Give 'n' boolean variables with value $x_1, x_2 \ldots x_n$, does at least one variable have the value "True"?

n = 4

ex (T F F T) ✓ Yes, in O(n) time.

Given 'n' integers $i_1, i_2, i_3 \ldots i_n$ is $\max(i_1, i_2 \ldots i_n) > 0$

(-30, 10, 0, 2) > 0

if any one find (+ve), we say Yes else no.

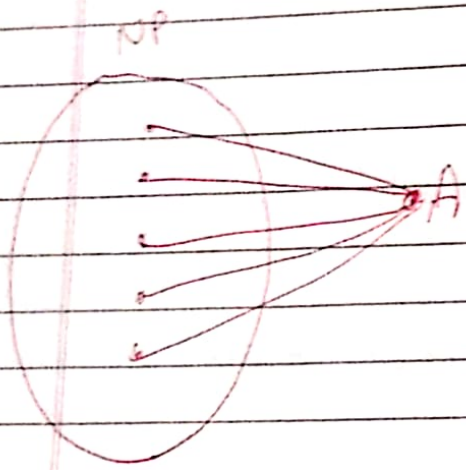
A → B

(T F F T)

↓ ↓ ↓ ↓

(1 0 0 1), atleast one '1'

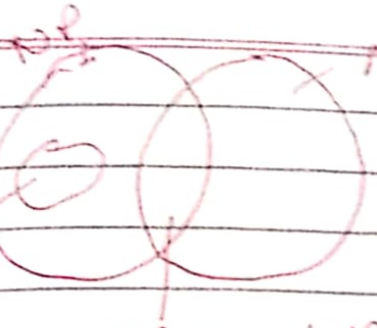
O(n) time. Hence B is easy so A is easy.



NP Hard: If every problem in NP can be polynomial time reducible to a problem 'A', then 'A' is called NP Hard. (A is outside the NP circle).

If 'A' could be solved in polynomial time, then every problem in NP is 'P'.
Hence it can be proved P = NP, but till now we have to proved it.

NP Complete: If it is in 'NP' & 'NP Hard' A is itself inside circle.



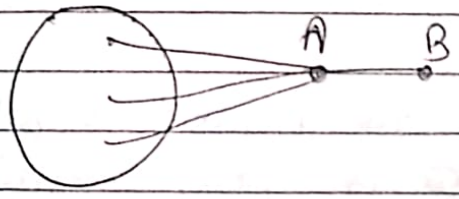
Many scientist believed that, but it is also yet to proved

NP Hard or NP complete is solved in polynomial time, then NP = P

NP problem or NP-complete is proven to be not solvable in polynomial time, then we can conclude that P ≠ NP

Status of NP is unknown.

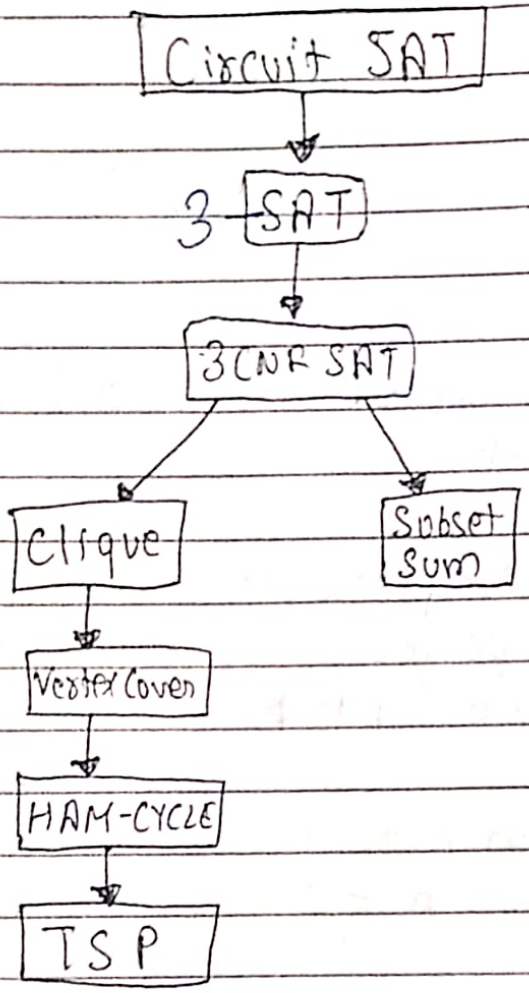
A is NP Hard & A $\xrightarrow[\text{"P"}]{}$ B, the 'B' is NP-Hard.



If A is NP Hard & B is NP. & A $\xrightarrow{\text{"P"}}$ B then B is NP complete.

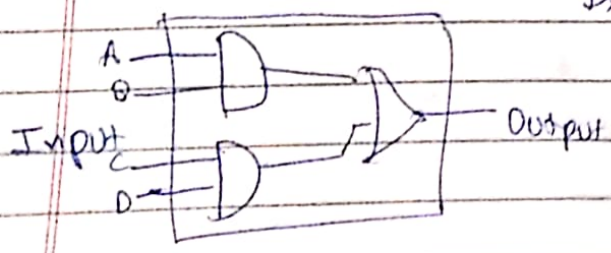
All are NP - problems - NP Complete problems

Exponential time we assumed to solve algo, with worst possible i/p.



Circuit SAT: We can make an algorithm which can say true ~~false~~ in polynomial time

Is there any combination possible of input, which say true.

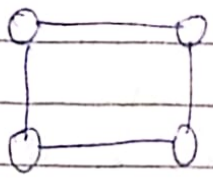


SAT:

$(x+y+z)\cdot(y\,z)\cdot(x+z)$

It is ~~inconsistent~~ also take individual calculation to solve with; which can take exponential time.

Vertex vector: How many watchman need to cover all the road. It take exponential time to find.



* RAM & SHYAM have been asked to show that a certain problem π is NP complete. RAM shows a polynomial time reduction from the 3-SAT problem to π, & Shyam shows a polynomial time reduction from π to 3-SAT. What is π.

3SAT $\xrightarrow{\text{poly}}$ π

↓

NPC

complete

$\pi \xrightarrow{\text{"P"}}$ 3-SAT (This is not examining any thing.

We can conclude that π is NP hard, because it converted in polynomial time. It is not NPC because it is not given that π is NP.

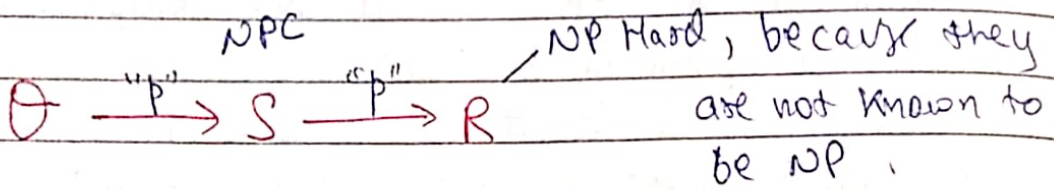
* The problem 3-SAT & 2-SAT are

3 SAT - $(a+b+c)$ - NP complete

2SAT = $(a+b)$, - P

If it is solved in polynomial time, then other NP will also solve.

* Let "S" be an NP complete problem & 'Q' & 'R' be two other problem not known to be in NP. Q is polynomial time reducible to 'S' & 'S' is polynomial time reducible to R.



Can't say because it can be ~~correct in any form~~ present in either P or NP-Hard.

* Let 'A' be a problem that belongs to the class NP. Then which of the following is True?

a) There is no polynomial time algo for 'A'.
False because Prime is NP, & is there exist poly algo to solve it

b) If A can be solved in ~~deterministically~~ in polynomial time, then P=NP
~~False~~, If A is prime, it doesn't mean all P = NP.

c) If A is NP-HARD, then it is NP complete.

d) It may be undecidable.

NP.

running algo but known its verification polynomial algo.

(5) Which of the following is true:

1) The problem of determining whether there exist a cycle in an undirected graph is in 'P'.
True. $O(v^2)$. While solving Kruskal, we do the same.

2) The problem of determining whether there exist a cycle in an undirected graph is 'NP'
Yes, because every P is NP.

3) If a problem 'A' is NP-complete, there exists a non-deterministic polynomial time algorithm to solve 'A'.
Yes, because non-deterministic mean it can go into various state in single step.

* Which of the following is not NP-Hard?
a) Hamilton circuit problem.
b) 0/1 Knapsack problem
c) finding bi-connected components of a graph.
d) The graph colouring problem.

* Let SHAM3 be a problem of finding hamilton-cycle in a graph $G = (V, E)$ with $|V|$ divisible by 3 & DHAM3 be a problem of determing if hamilton cycle exist in such graph.
Which of the following is true

* DHAM → NPC
SHAM → $|V| \cong 0 \bmod 3$ (If it is NP, because we provide sol^n & verifier weather verifier is divisible by 3 or not).

polynomial HW.

DHAM $\xrightarrow{\text{"p"}}$ SHAM

If DHAM is Hard then SHAM is also hard & it will NP complete.

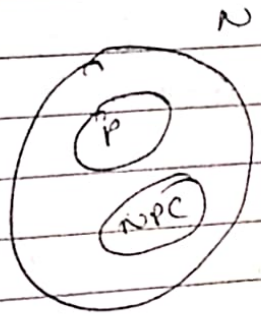
* We convert every problem into SHAM.

DHAM produce 3 result, remainder
0 → This will directly match
1 → We split one vertex into two & similarly do for
2 → We split one vertex into two.

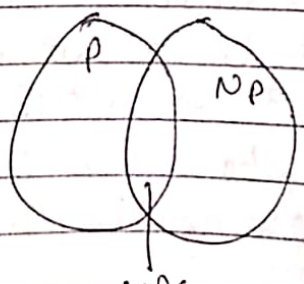
Now we have converted DHAM into SHAM, so it is NP complete.

* Suppose a polynomial time algorithm is discovered directly compute the largest clique in the given graph. In this scenario which one of the following represent the corresponding venn diagram of the complexity classes P, NP & NPC?

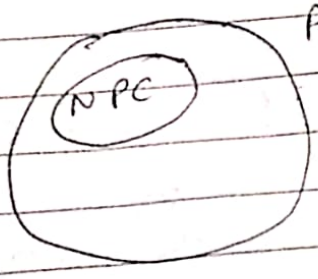
a)



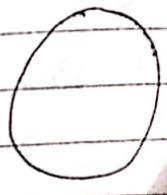
b)



c)



d)



time, then we can solve every problem of NP in Polynomial time. NPC are real hero or don.

Because every problem can be reduced to clique problem & then we get polynomial time algo. But NPC will be inside because we can match any problem with other problem.

Q. Consider the decision problem 2CNF-SAT
a) NP complete
b) Solvable in

* Consider the decision problem "P" 2CNF-SAT
a) NP complete
b) Solvable in polynomial time by reduction to directed graph reachibility.
c) Solvable in constant time since any input instance is satisfiable.
d) NP-Hard, but not NPC

One more

* If an NP-hard problem can be solved in polynomial time, then P=NP (T/F)

True, because every problem in NP can be reduce to NP hard in polynomial time & NP-Hard itself solve in polynomial time

* The set of NP problem is not closed under polynomial time reduction (T/F)

True, it is not closed polynomial time reduction