Algorithms

Algorithms are informal description of any problem in Computer Science. Which we write before writing any program in any language like C, C++, Python.

Also we can say that algorithms are blueprint of that program. Which we will write to solve problem.

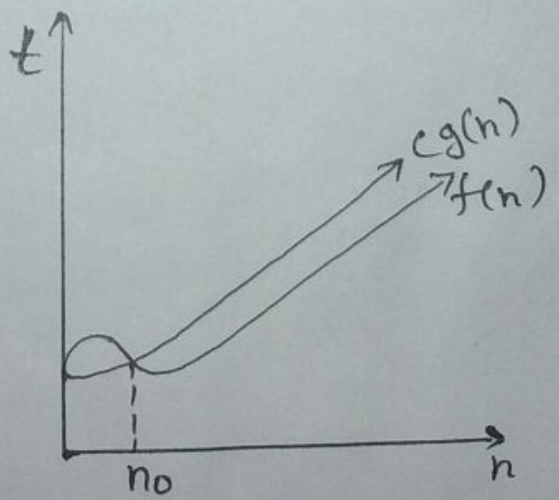
For a problem P we can have many algo. such as A_1, A_2, A_3 ---.

We analyse Algo. in terms of Time & space complexity. before Implementing it in a program.

Algo. which have less time & space complexity will use in program.

Asymptotic Notation

* Big (on):-



Conditions

$f(n) \leq cg(n)$

$n \geq n_0$

$c > 0, n_0 \geq 1$

$\boxed{f(n) = O(g(n))}$

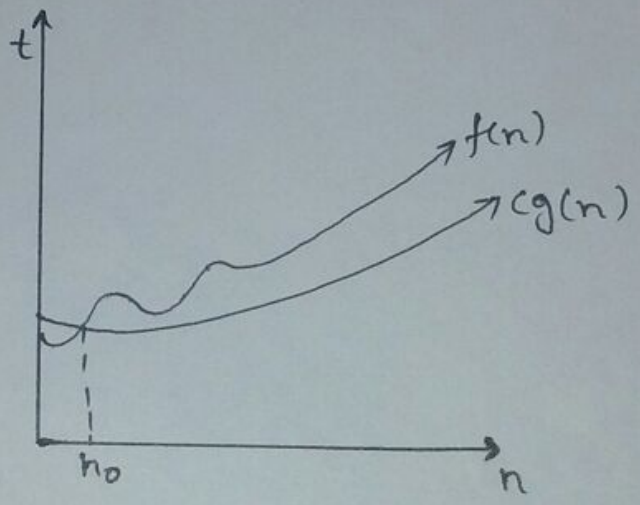
~~n=s~~

n is the size of Input.

f(n) Here f is the function of n.

→ For a given function f(n) if it can be bound with some other function c g(n) in such a way that after some input n_0 value of c g(n) always greater than f(n).

Big omega (Ω):-



Conditions!-

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

$n \geq n_0$

$c > 0, n_0 \geq 1$

For a given function f(n) after some value of n (i.e n_0) function cg(n) will always less than f(n).

Ex:-

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

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

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

$3n+2 \geq cn$ ($c=1$), ~

$n_0 \geq 1$ ~

Big theta (Θ) :-

Condition:

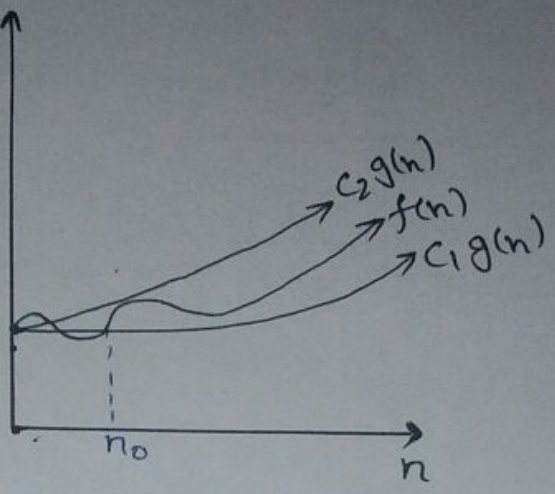
$f(n) = \Theta(g(n))$

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

$c_1, c_2 > 0,$

$n \geq n_0,$

$n_0 \geq 1$



if a given function f(n) is bonded by functions $c_1 g(n)$ and $c_2 g(n)$. then we can say f(n) is theta of g(n).

Ex:-

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

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

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

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

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

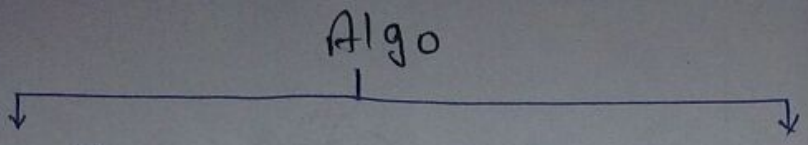
Why Notations:-

O :- It gives worst case. It means in any case Comblexity (time) will not exceed this.

Ω :- It gives best case.

Θ :- It gives average case. For some algo. Worst & best case are same then we use it.

Algo Type!-



Iterative

⇒

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

Recursive

⇒

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

* Iterative & Recursive both are equivalent. Iterative can be converted into recursive & recursive can be converted into Iterative.

Recursive!-

If there is a function A which takes an input of size n & it is again computing the value in terms of same function. called recursive.

Iterative!-

If there is no function calling. but it having loops like for loop, while loop then called Iterative.

Time Complexity :-

It is the time taken by the program to execute completly.

Time Complexity of Iterative Programs (functions) :-

Ex1

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

Time Complexity of this program will O(n) Because for loop will run till n elements.

Ex2

```
{
int i, j;
for(i=1 to n)
for( j=1 to n)
Printf("workforwin");
}
```

Time Complexity = $O(n^2)$

Here first loop run n times & second loop will also run for n times. So workforwin will print n^2 times. & its time complexity will also be n^2 times.

Ex 3

```
A()
{
int i, j, k;
for(i = n/2; i <= n; i++)          — n/2
for(j = 1; j <= n; j = 2 * j)      — log2n
for(k = 1; k <= n; k = k * 2)      — log2n
Printf("work for win")
}
```

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

Soln:-

first for loop will run $n/2$ times — ①

second for loop will run $\log_2 n$ times — ②

third for loop will run $\log_2 n$ times — ③

Multiply ~~Add~~ ① ② ③ → $n/2(\log_2 n)^2$

→ $O(n(\log_2 n)^2)$

Recursive Program (function):-

In order to ~~solve~~ finding recursive program time complexity first we have to write recursive eq of a program.
Then after writing eq. we have many way of solving that recursive equation. All methods are ~~de~~ discussed below one by one!-

1) Back substitution:-

In back substitution method we substitute value of function in recursive eq. till End of the function value.

Ex:-

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

Time taken to solve A(n) is T(n).
then time taken to solve (n-1) will be T(n-1)
Lets write recursive eq. for above ~~pro~~ algo.

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

* Here 1 is constant we can take any constant like c, 1 ...

Apply Back substitution on Eq.

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

⋮

Substitute Eq 2 in Eq 1.

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

$= 2 + T(n-2)$

~~$= 2 + T$~~ ⇒ Substitute Eq 3.

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

$= 3 + T(n-3)$

!

!

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

Stoping condition of recursion

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

$= 1$; $n = 0$

$\Rightarrow$ $n - k = 1$

$k = n - 1$

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

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

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

Time Complexity $O(n)$

* one drawback of this method is it is slow.

inly the recursion tree method is used with divide & conquer algo.

In recursion tree we have one root & it have child node. Here each node represent cost.

At each level we calculate cost or complexity.

Then at last add all level cost & after solving we have final cost of algorithm.

3) Masters Theorem :-

Masters theorem applies on recursive relations (functions). It is fastest way of getting the solution as compare to ~~recursion~~ recurrence tree and back tracking. It is derived from recurrence tree.

For a ~~recurrence~~ recursive function

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

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

We have three cases :-

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

2) if $a = b^k$
   a) if $p > -1$, then $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$
   b) if $p = -1$, then $T(n) = \Theta(n^{\log_b a} \log\log n)$
   c) if $p < -1$, then $T(n) = \Theta(n^{\log_b a})$

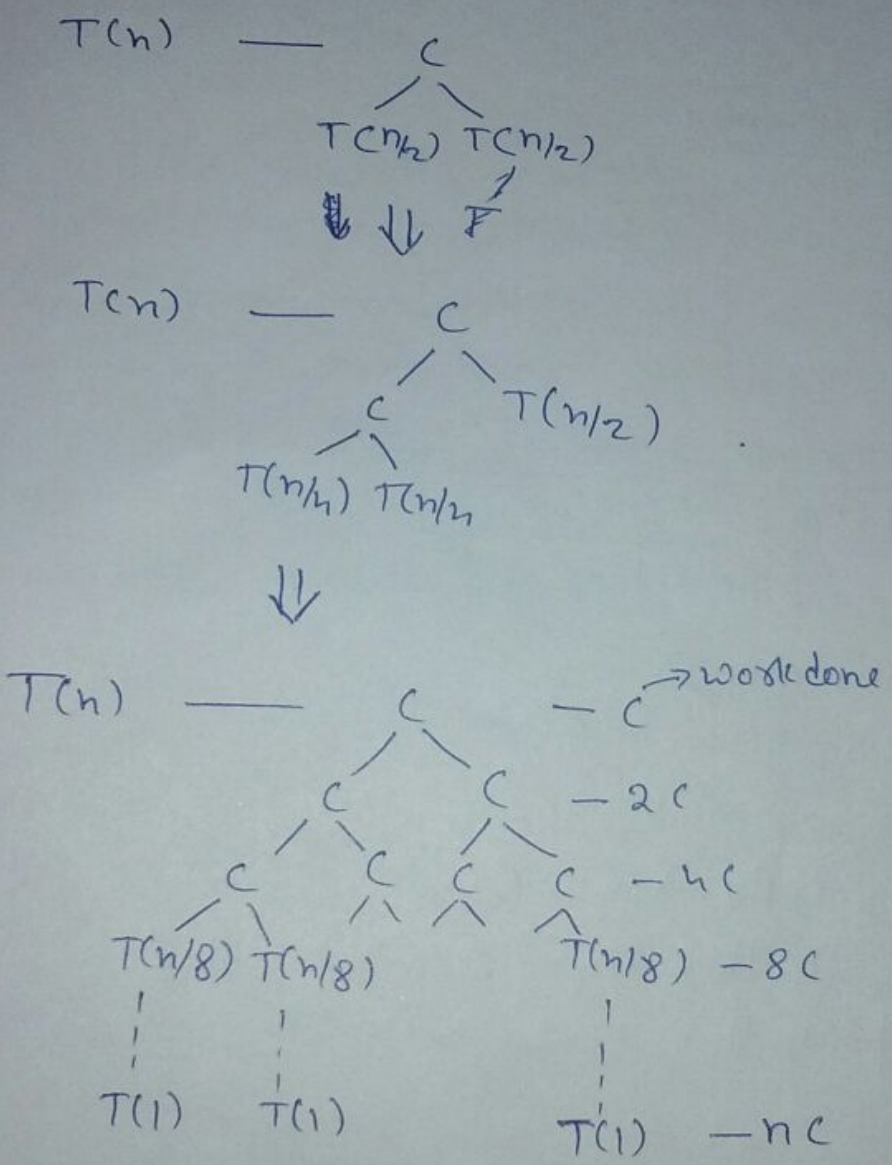
3) if $a < b^k$
   a) if $p \geq 0$, then $T(n) = \Theta(n^k \log^p n)$
   b) if $p < 0$, then $T(n) = O(n^k)$

2) Recursion Tree Method!-

Ex Let we have recursive Eq.

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

Sln!-



$$c + 2c + 4c + \cdots nc$$
$$c(1 + 2 + 4 + \cdots n)$$

assume $n = 2^k \Rightarrow c(\underbrace{1 + 2 + 4 + \cdots 2^k}_{G.P})$

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

Time Complexity = $O(n)$

Space Complexity for Iterative programs:-

```
Algo(A,1,n)
{
 int i;
 for(i=1 to n)
 A[i]=0;
}
```

Calculation of space complexity:-
Size of array is n elements.
So in entire program for n size input we take only one variable that is i.
So its constant.
∴ Space Complexity = O(1)

Space Complexity for recursive program:-

```
A(n)
{
if(n≥1)
{
 A(n-1)
 Pf(n);
}
}
```

Here for A(n) computing (n+1) recursive call occurs.
& for each call we need k cell in memory
So space required (n+1)k = O(n)

Ex:-

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

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

Check a & b^k condition

$a = 3, \ b = 2$

$a \leftarrow (3) \qquad (2^2) \rightarrow b^k$

$3 < 4$

So apply masters theorem case 3.

$a < b^k$

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

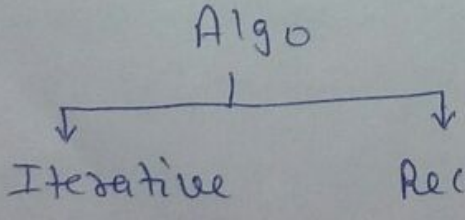
After applying this condition we get

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

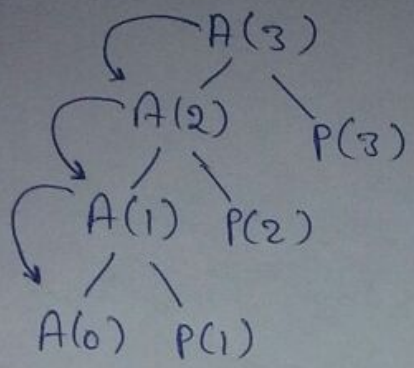
$\boxed{T(n) = \Theta(n^2)}$

Space Complexity

for given a program P how many memory cells are required in order to finish the algorithm or program. This analyse in terms of given input size.



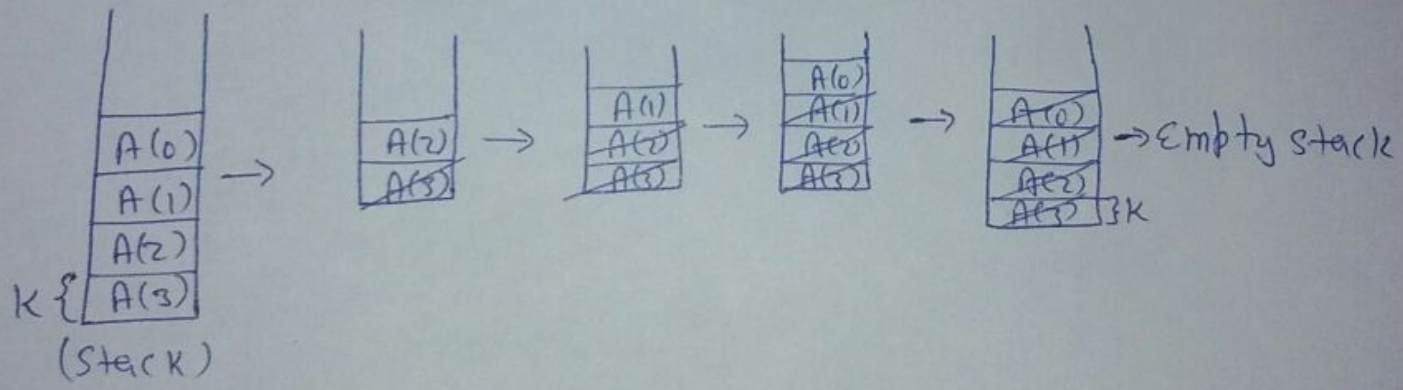
for above program let value of n = 3 then



for A(3) there is 4 recursive call occurs.
Here Printf call for 3 times these are P(3), P(2), P(1)
At A(0) condition will not satisfy. So execution of program will stop here.

Space required :-

Here we use stack for PUSH & POP of elements



first A(3) will PUSH in stack before Pushing A(3) stack was empty. When Printf will call after PUshing element ~~em~~ element will be printed. once element ~~to~~ got print POP A(3) from stack again stack will empty. Then PUSH A(2) in stack & again same process will call. and so on till last element.
For each element in stack we required k ~~et~~ cell. So Here After performing operation we free element from stack So for all elements we need only k cell in stack.

Sorting Techniques

~~Insertion Sort Algo~~ :-

Sorting :-

Sorting is arranging the elements either in ascending or in descending order. For this we use some algorithms. In Computer science we have different types of sorting algorithms.

Ex. of sorting :-

Workforwin → ~~fiknoo~~

fiknoorrww

Input

output.

Ex. 2

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

Input

output.

Algo

iterative

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

Simple like this loop/statement then iterative

Recursive

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

function Calling means recursive.

* If a Algo is not iterative or recursive then time complexity will O(1).

* T.C. of Iterative Programs

Ex

```
A()
{
int i,j;
for (i=1 to n)
for (j=1 to n)
Pf("Google");
}
```

$O(n^2)$

```
A()
{i=1, s=1;
while (s<=n)
{i++
s=s+i;
Pf("Ankit");
}
}
```

	1	2	3	4	5	6		(>n)
s	1	3	6	10	15	21	--	
i	1	2	3	4	5	6	--	

(K) → stop loop. value will be more than n. let that time is k.

after iteration (k) → loop stop. → value of s will $\frac{k(k+1)}{2}$

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

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

Ex

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

$i <= \sqrt{n}$

$O(\sqrt{n})$

```
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("Ankit");
}
}
}
```

Sol^n

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

i = n
j = n
k = n×100 times

$= 100 + 2\times100 + 3\times100 \text{ --- } n\times100$

$= 100(1+2+3 \text{ --- } n)$

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

$= O(n^2)$

→

Ex

```
A()
int i,j,k,n;
for(i=1; i<=n, i++)
{
for(j=1; j<=i^2; j++) → dependent j
{
for(k=1; k<= n/2; k++) → independent
{                        loop not involve
 Pf("Google");           i or j (not depend
}                        on i/j)
}
}
```

Soln

i = 1	i = 2	i = 3
j = 1 time	j = 4	j = 9
k = n/2 * 1	k = n/2 * 4	k = n/2 * 9

$i = n$ $j = n^2$ $k = n/2 * n^2$

$= n/2 * 1 + n/2 * 4 + n/2 * 9 --- + n/2 \times n^2$

$= n/2 (1 + 4 + 9 + --- + n^2)$

$= n/2 \left(\frac{n(n+1)(2n+1)}{6} \right)$ → sum of all first n natural no.

$= O(n^4)$

Key

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

Ex- A()

```
{for(i=1; i<n; i=i+2)
 Pf("CEO");
}
```

$i = i+2 \rightarrow O(\log_2 n)$

$i = i+3 \rightarrow O(\log_3 n)$

$i = i+4 \rightarrow O(\log_4 n)$

Soln

$i = 1, 2, 4 \quad --- (n)$

$2^0, 2^1, 2^2 \quad --- (2^k)$ stop

$2^k = n$

$k = \log n$

$O(\log n)$

Ex-

```
{A()
 int i,j,k;
n/2 ← for(i=n/2; i<=n; i++)
n/2 ← for(j=1; j<=n/2; j++)
log2n for(k=1; k<=n; k=k*2)
        Pf("Ankit");
}
```

Soln

$n/2 * n/2 * \log_2 n$

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

```
A()
{
int i,j,k;
for(i=n/2; i<=n; i++) → n/2
 for(j=1, j<=n, j=2*j) → log2n
 for(k=1; k<=n; k=k*2) → log2n
  Pf("CSE");
}
```

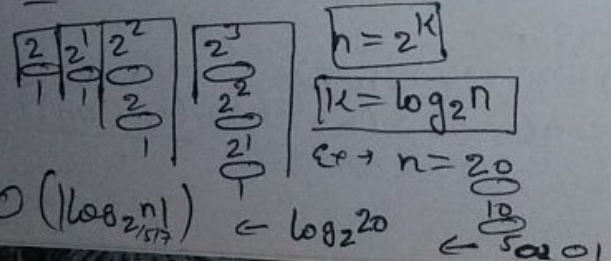
Soln $n/2 (\log_2 n)^2$

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

Ex → Assume $n \geq 2$

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

Soln



Ex- A()

```
{
for(i=1; i=n; i++) → n
for (j=1; j<=n; j=j=j+i)
Pf("Ankit");
}
```

Soln

i = 1	i = 2	i = 3	i = k
j = 1 to n	j = 1 to n	j = 1 to n	j = 1 to n
n times	j = 1,3,5,7,9 --	n/3 times	n/k
j = 1,2,3,4,5 --	n/2 times		

$i = n$, $j = 1$ to n, n/n

$= n + n/2 + n/3 + \cdots n/n)$

$= n(\log n) \rightarrow n(1 + 1/2 + 1/3 + \cdots 1/n)$

$= O(n \log n)$

Ex- A()

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

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

$n*(k+1) = n(\log\log n + 1)$

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

$\Rightarrow \log\log n = k$

$n*(k+1) = n(\log\log n + 1)$

$O(n \log\log n)$

* Recursive function Time Complex

A(n) — T(n) time taken
{
if (—) → constant
return (A(n/2) + A(n/2))
}
constant

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

Ex A(n) → T(n)
{
if (n>1) → constant
return (A(n-1) → time taken T(n-1)
}

Sln

recursive eqn. will

$T(n) = 1 + T(n-1)$ — ①

Back substitution method

to solve it

$T(n-1) = 1 + T(n-2)$ — ②

$T(n-2) = 1 + T(n-3)$ — ③

from eq ② using in ①

$T(n) = 1 + 1 + T(n-2)$
$= 2 + T(n-2)$
$= 2 + 1 + T(n-3)$
$= 3 + T(n-3)$
⋮
stop when $k + T(n-k)$ → ①

$T(n) = 1 + T(n-1)$; $n > 1$
$= 1$ → stop ; $n = 0$

① when stop

$n - k = 1$
$k = (n-1)$

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

$O(n)$

Ex Recursive eq of a program is

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

find time comply.

Sln

$T(n-1) = (n-1) + T(n-2)$ — ②

$T(n-2) = (n-2) + T(n-3)$ — ③

② in ① then

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

→ continue

$n-(k+1) = 1$

$n-k-1 = 1$

$\Rightarrow \boxed{k = n-2}$ ✓

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

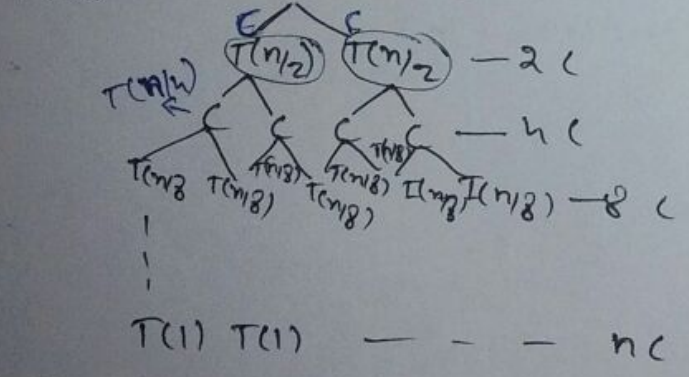
$= n + (n-1) + (n-2) + \cdots 2 + 1$ → sum of natural no

$= \frac{n(n+1)}{2} = O(n^2)$ ✓

(*) Key $\boxed{(n+k)^m = O(n^m)}$ ✓

★ Recursion tree method (* interview *)

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



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

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

*→ assume $n = 2^k$ → not effect on ans. help to solve eqn.

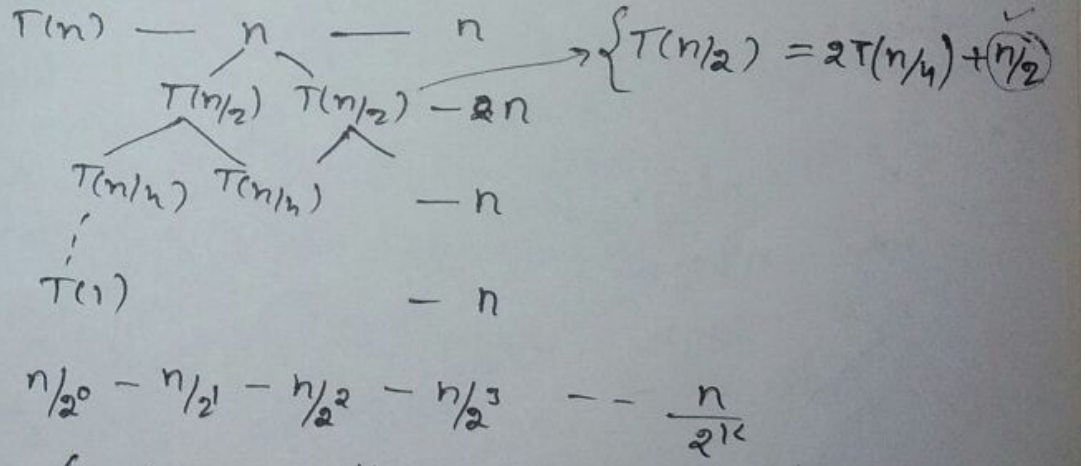
~~c+2c~~ G.P

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

$c\left(\frac{1(2^{k+1}-1)}{2-1}\right)$ $\left(\frac{a \cdot r^{(k+1)} - 1}{r-1}\right)$ → total terms

$= c(2^{k+1} - 1) = c(2n-1) = O(n)$ ✓

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



$n/2^0 - n/2^1 - n/2^2 - n/2^3 \cdots \frac{n}{2^k}$

$(k+1)$ $\quad n = 2^k$

↓ $\quad k = \log n$

$n(\log n + 1)$

$= O(n \log n)$ ✓

* Comparing various function to analyse time complexity.

Ex 2^n n^2 → which is bisser

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

n $2 * \log n$

2^{100} $2 * \log 2^{100}$

bisser 200

2^{100}, 2^{200}, 2^{300}

Ex 3^n 2^n

$\cancel{\log 3}$ $\cancel{\log 2}$

bisser

Ex n^2 $n \log n$

$n * \cancel{n}$ $\cancel{*} \log n$

n bisser $\log n$

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

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

$n = 2^{128}$

128 $100 * \log \log 2^{128}$

$\log 2^{128}$ → 128 → 7

700

$n = 2^{1024}$

1024 bisser

$100 * \log \log 2^{1024}$

$\log 2^{1024}$ → 1024 → 10

1000

Ex $n^{\log n}$ $n \log n$

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

$n = 2^{1024}$ 1024

$1024 * 1024$ $(1024 + 10)$

$n = 2^{2^{20}}$

$2^{20} * 2^{20}$ $2^{20} + 20$

bisser

$f(x) = O(g(x))$ (bisser) after cancelling ~~equal~~ constant + say order

$\sqrt{\log n}$	$\log\log n$
$\frac{1}{2}\log\log n$	$\log\log\log n$
$n = 2^{2^{10}}$	$\log\log\log 2^{2^{10}}$
$\frac{1}{2}\log\log 2^{2^{10}}$	$\log 10$
5	3.5

Ex

$n^{\sqrt{n}}$	$n^{\log n}$
$\sqrt{n}\log n$	$\log n \log n$
$\sqrt{n}$	$\log n$
$\frac{1}{2}\log n$	$\log\log n$
$n = 2^{128}$	
$\frac{1}{2}\log 2^{128}$	$\log\log 2^{128}$
$= 64$	$= 7$
bigger	

$2^{256} \to 7$
$2^{128} \to 7$
$2^{1024} \to 10$

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

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

Sln

	0–99	100–9999	10000 ---
$f(n)$	n^3	n^3	n^2
$g(n)$	n	n^3	n^3

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

Ex

$f_1 = 2^n$, $f_2 = n^{3/2}$, $f_3 = n\log n$

$f_4 = n^{\log n}$

2^n	$n^{3/2}$
$n\log 2$	$\frac{3}{2}\log n$
n	
$n = 2^{128}$	$\frac{3}{2}\log 128$
2^{128}	3×64

2^n	$n\log n$
$n\log 2$	$\log n + \log\log n$
n	$\log n + \log\log n$
$n = 2^{128}$	$128 + \log\log_2 128$
2^{128}	$128 + 7$

$f_1 > f_4 > f_2 > f_3$

2^n	$n^{\log n}$
$n\log 2$	$\log n \log n$
$n = 2^{128}$	
2^{128}	128×128
	$2^7 \times 2^7$
bigger	2^{14}

then step-2.

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

$n^{\log n}$	$n\log n$
$\log n \times \log n$	$\log n + \log\log n$
$n = 2^{128}$	
128×128	$128 + 7$

step-3

$n^{3/2}$	$n\log n$
$\frac{3}{2}\log n$	$\log n + \log\log n$
$n = 2^{128}$	
$\frac{3}{2} \times 128$	$128 + 7$
$\frac{3}{2} \times 1024$	$1024 + 10$

* Masters Theorem (Imp)

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

if this condition is fulfill in question then apply this theorem

$a \geq 1$, $b > 1$, $k \geq 0$ & p is real no

$\log_b a$

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

2) if $a = b^k$
   a) if $p > -1$, then $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$
   b) if $p = -1$, then $T(n) = \Theta(n^{\log_b a} \log\log n)$
   c) if $p < -1$, then, $T(n) = \Theta(n^{\log_b a})$

3) if $a < b^k$
   a) if $p \geq 0$, then $T(n) = \Theta(n^k \log^p n)$
   b) if $p < 0$, then $T(n) = \Theta(n^k)$

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

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

$a \quad b^k$

$3 \quad 2^2$

$3 < 4$

3(a) condition abbliyed here

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

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

$= \Theta(n^2)$ ✓

$(\log n)^2$	$\log^2 n$
↓	
$\log n * \log n$	$\log\log n$

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

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

$a \quad b^k$

$4 = 2^2$

2)(a) condition

$T(n) = \Theta(n^{\log_2 4} \log^0 n)$

$= \Theta(n^2 \log n)$ ✓

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

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

$1 < 2^2$

3(a) condition $T(n) = \Theta(n^2 \log^0 n)$

$= \Theta(n^2)$ ✓

Ex $T(n) = 2^n T(n/2) + n^n$ × Theorem no applyed

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

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

$16 > 4^1$

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

$= \Theta(n^{\log_4 16})$ → 2

$= \Theta(n^2)$ ✓

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

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

$2 = 2$

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

$= \Theta(n^1 \log^2 n)$ ✓

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

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

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

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

$= \Theta(n \log\log n)$ ✓

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

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

$2 < 4^{0.51}$

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

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

$= \Theta(n^{0.51})$ ✓

Ex $T(n) = 0.5T(n/2) + 1/n$

$a = 0.5$ ✗ → can't solve this using theorm condition not full fill.

Ex $T(n) = 6T(n/3) + n^2 \log n$

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

$6 < 3^2$

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

$= \Theta(n^2 \log^1 n)$ ✓

Ex $T(n) = 64T(n/8) = n^2 \log n$ ✗ Wrong — come divide sitin so can't solve using m.T.

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

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

$7 < 3^2$

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

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

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

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

$4 > 2^0$

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

$= \Theta(n^2)$ ✓

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

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

$\sqrt{2} > 2^0$

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

$= \Theta(n^{\log_2 \sqrt{2}}) = \Theta(\sqrt{n})$ ✓

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

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

$2 > 2^{1/2}$

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

$= \Theta(n)$ ✓

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

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

$3 > 2$

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

$= \Theta(n^{\log_2 3})$ ✓

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

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

$3 > 3^{1/2}$

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

$= \Theta(n^{\log_3 3})^{-1} = \Theta(n)$ ✓

Ex $T(n) = 4T(n/2) + n^1$

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

$4 > 2$

$T(n) = \Theta(n^2)$ ✓

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

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

$3 < 4^1$

$T(n) = \Theta(n \log n)$ ✓

Analysing space Complexity of Iterative & Recursive algorithms

Also
- Itr.
- Rec.

Ex
```
Algo(A,1,n)        // n → size for array not count because it is fix size
{
int i,j=10;        // 2 → constant
for(i=i to j)
A[i]=0;
}
```
O(1)

Ex
```
Algo(A,1,n)
{
int i;             // 1
create B[n];       // n+1 → n
for(i=1 to n)
B[i]=A[i];
}
```
O(n)

Ex
```
Algo(A,1,n)                 // n² → n²+2
{
create B[n,n]               // 2
int i,j;
for(i=1 to n)
for(j=1 to n)
B[i,j]=A[i];
}
```
O(n²)

S.C for Recursive ~~fct~~ also.

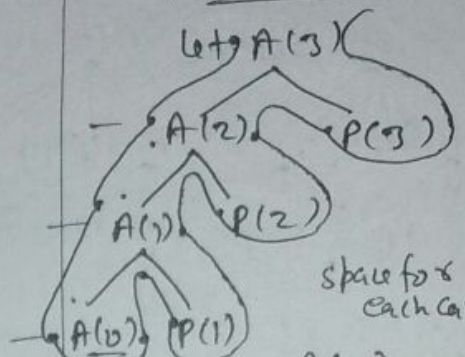
1) Head R. algo → function call in head
2) Tail/Under/bottom R. algo → function call in last after Printf.
3) Body R. also → function call anywhere.

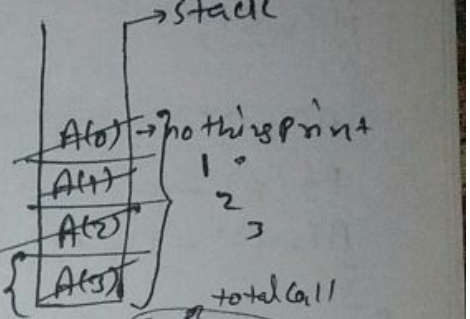
Ex →
```
A(n) → (n+1)
{
if (n≥1)
{
A(n-1);        // funct call at head
Pf(n);
}
}
```

Ex Soln. Let A(3)

Stack method

Tree method for short algo.





A(3) — 4
A(2) — 3
A(n) — A(n-1)

O(n)

T.C. of same algo → Here ~~m.t~~ will not applicable

$T(n) = T(n-1) + 1 \quad ; n=0$
$\qquad\qquad ; n \geq 1$ → {Back method use}

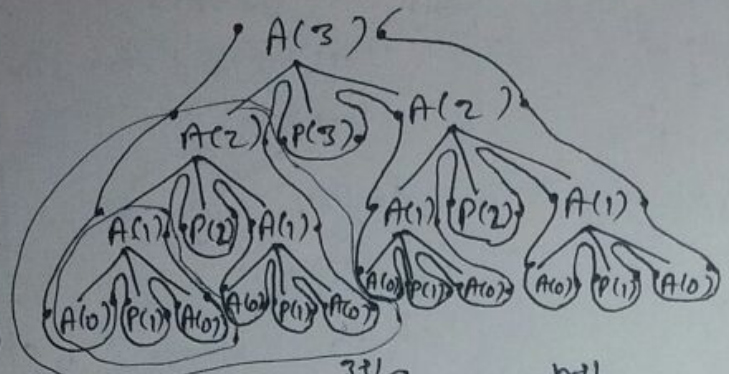
1 → constant it can cork any no
Pf take this time.

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

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

Ex

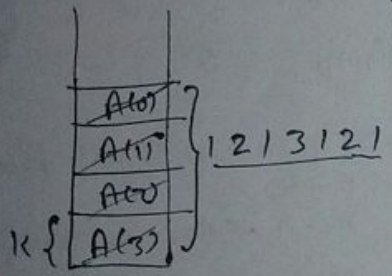
```
A(n) - T(n)
{
if(n≥1)
{
A(n-1); -T(n-1)
Pf(n) -1-(
A(n-1); -T(n-1)
}
}
```



$A(3) = 15 \quad 2^{3+1}-1$ $\quad A(n) = 2^{n+1}-1$

$A(2) = 7 \quad 2^{2+1}-1$

$A(1) = 3 \quad 2^{1+1}-1$



$A(n)$

$(n+1)k$

$O(nk)$

Space complexity $O(n)$ ✓

Time complexity

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

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

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

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

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

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

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

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

$= 2^nT(0)+2^{n-1}+2^{n-2}+---2^2+2^1+1$

$= \underbrace{2^n+2^{n-1}+2^{n-2}+--2^2+2+1}_{G.P}$

$= O(2^{n+1}) \quad = O(2^n)$ ✓

T. complexity

* To reduce ~~space~~ time complexity use Dynamic programming

In Dynamic Programming we use result again if same process (step) is repeting.

Ex→ Here A(2) is printing 121 & again A(2) printing 121 so we use result we get first after solveing A(2) & A(1) is also repeting. we store value in a table form to use it again.

3	2	1	0

123k(21) ① Ex3 n+1

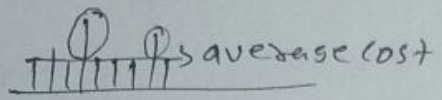
Amortized Analysis →

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

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

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

→ Techiques used in amortized analysis
- Aggregate method
- Accounting method
- Potential method



→ Aggregate Analysis! –

Hash table insertions

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

$$\frac{T_1 + T_2 + T_3 \text{ --- } T_n}{n}$$

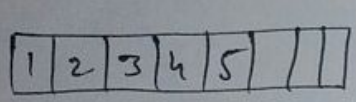
Insert 1 → (overflow) — If table is 0

Insert 2 (overflow)

Insert 3 (overflow)

Insert 4 (overflow)

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	16	---
Cost:	1	2	3	1	5	1	1	1	9	1	---

(1 moves, 1 insert) (2 moves, 1 insert) (4 moves, 1 insert) (8 moves, 1 insert)

The Amortized cost =

$= (1 + 2 + 3 + 1 + 5 + 1 + 1 + 1 + 9 + \cdots)$

$= (\underbrace{1 + 1 + 1 + \cdots + 1}_{n \text{ times}}) + (1 + 2 + 4 + 8 + \cdots)$

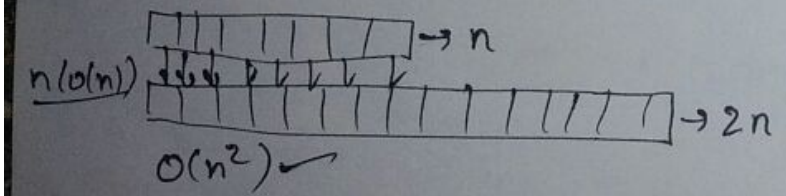
$1 + 2 + 4 + \cdots 2^{\lceil \log_2 n \rceil}$

$\lceil \log_2 n \rceil + 1$

$\frac{n + 2n}{n} = 3$ — constant value

∴ The amortized cost = $O(1)$ — constant.

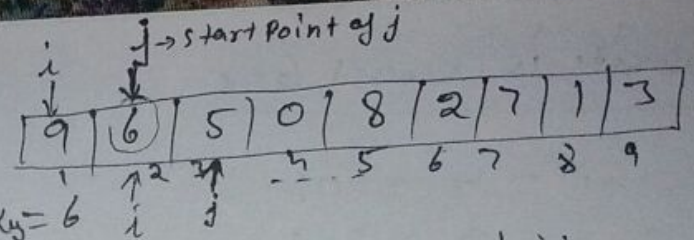
Worst value for n



n (O(n))

$O(n^2)$

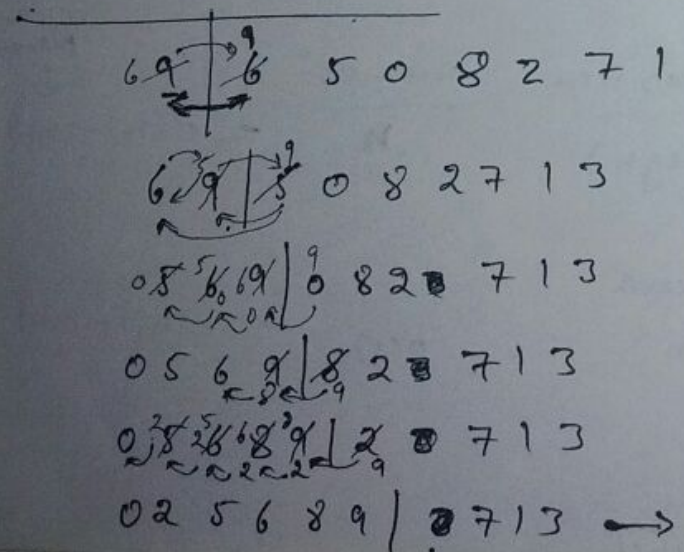
*Insertion short algorithm & analysis

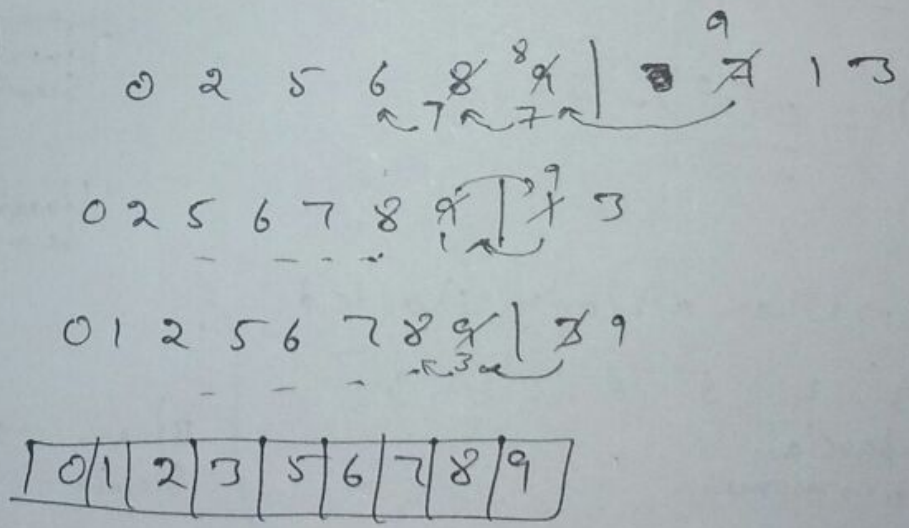



```
Insertion-Sort(A)
{
for j = 2 to A.length
 key = A[j]
 // insert A[j] into sorted sequence A[1 - j-1]
 i = j-1
 While ( i > 0 & A[i] > key)
  A[i+1] = A[i]
   i = i+1
 A[i+1] = key
}
```

Comparision mount

j = 2	1 + 1	= 2
j = 3	2 + 2	= 4
j = 4	3 + 3	= 6
⋮		
j = n	(n-1) + (n-1)	= 2n-2
		= 2(n-1)





Time complexity in worst case

2 — 1+1 = 2 → 2(1)
3 — 2+2 = 4 → 2(2)
4 — 3+3 = 6 → 2(3)
⋮
n — (n-1)+(n-1) = 2(n-1)

2(1) + 2(2) —— 2(n-1)

2(1+2 — — — n-1)

$= \frac{2(n-1)(n)}{2} = O(n^2)$ ═

Space complexity

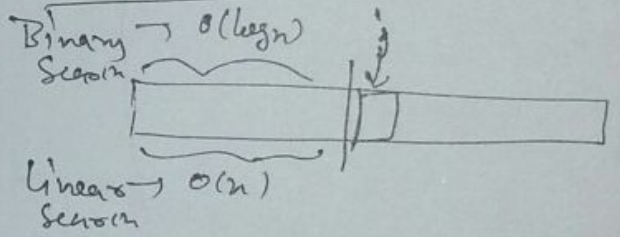
(key) (i) (j)

size of input — n

require only 3 variable

— O(1) ═

* Insertion short called Insertion algo.



Best Case → when already shorted

1 2 3 4 5 6 7 8 9

	comparision
j=1	1+0 → moment
j=2	1+0
j=3	1

n — (n-1)

Ω(n-1) ═

	Comparision	moment	
Binary Search →	O(log n)	n	= O(n) → O(n²)
double linked list →	linear search O(n)	O(1)	= O(n) → O(n²)

form element

* Time Complexity can't be reduce even by using B.S & d.LinkList.

Merge sort or out of space

* Better than Insertion sHort in time complexity

* Merge Algo :

MERGE (A, p, q, r) → p to q are sorted & q+1 to r is sorted

{

$n_1 = q - p + 1$ → no of element in 1st list

$n_2 = r - q$ → no of element in 2nd list

let L[1 ... n+1] and R[1 to n_2+1] be new array

for (i = 1 to n_1) → used to copy 1st list into array

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

for (j = 1 to n_2) → used to copy 2nd list into array

R[j] = A[q + j]

$L[n_1+1] = \infty$

$R[n_2+1] = \infty$

i = 1, j = 1

for (k = p to r)

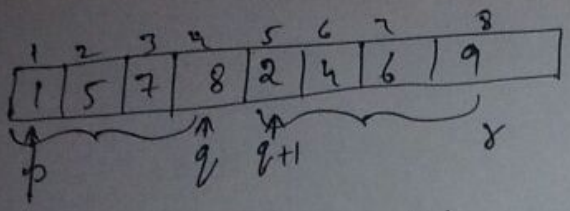
if (L[i] ≤ R[j])

A[k] = L[i]

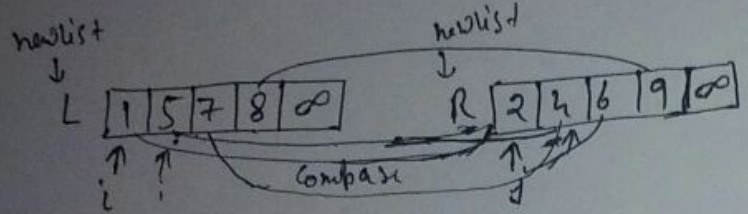
i = i + 1

else A[k] = R[j]

j = j + 1

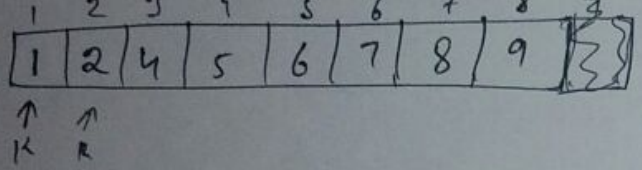


O(n) → space require for element.



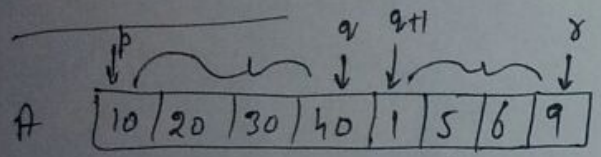
- O(n)

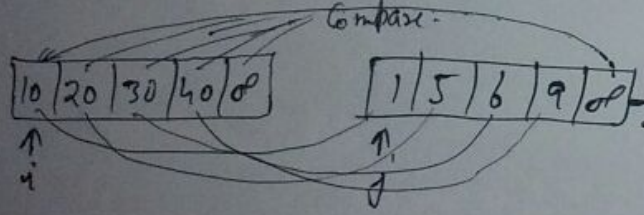
time complexity & space complexity



comparision ↓ - (n+n) = O(n)

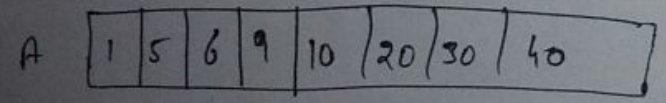
Copy.





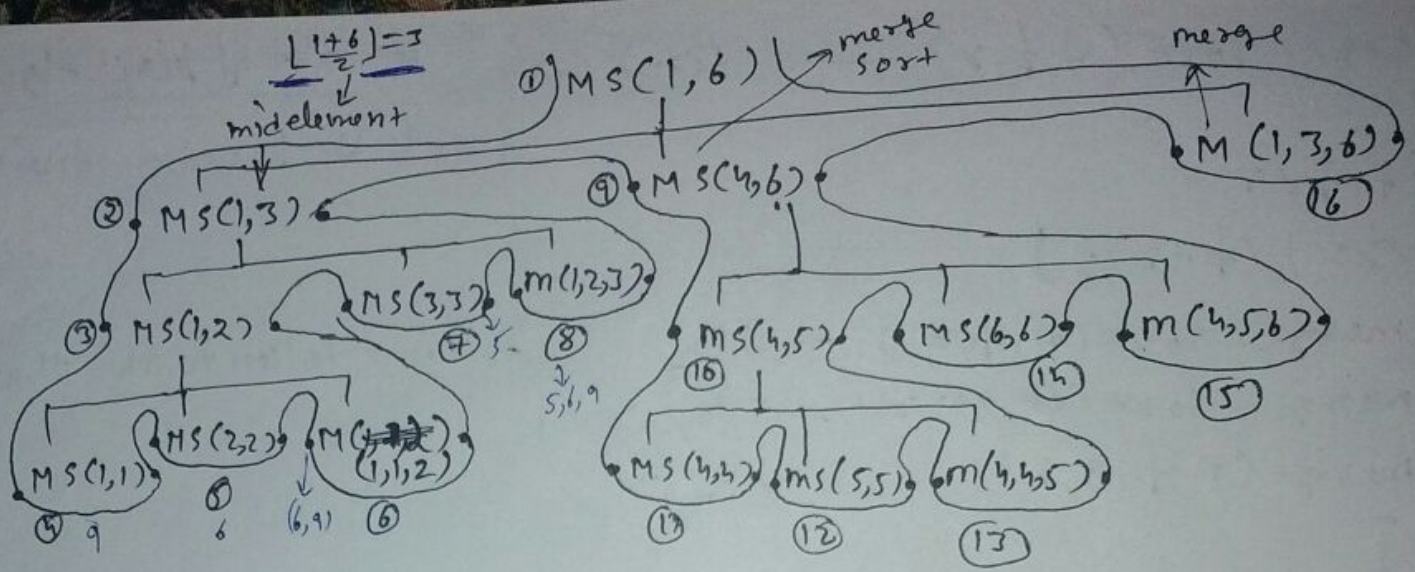
→ it will copy remaining list'd but in array thats why its resid.

→ this list is sorted that means 1st list is sorted d by comparing ∞ all element will but in array.

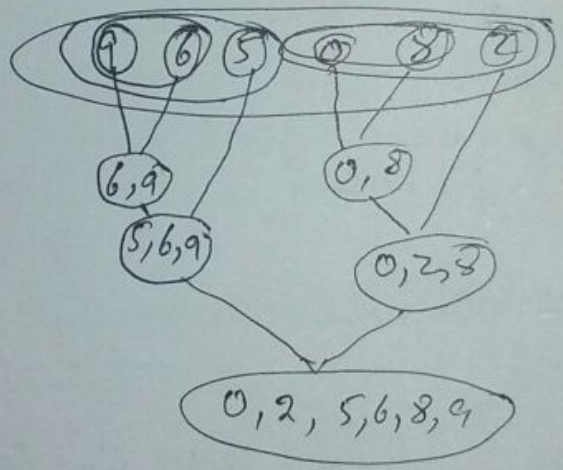


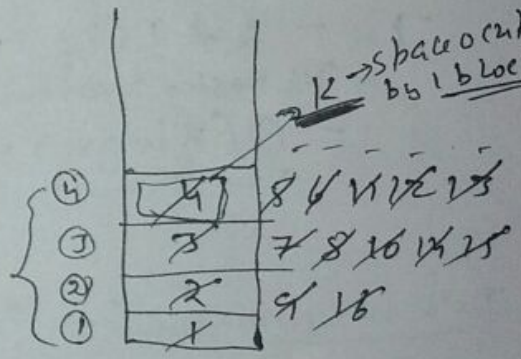
Sort Algo

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



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





Stack space — O(logn)

O(n) → space for merge

Total space require O(n+logn)

Space Complexity → O(n)

n → size of array

(⌈logn⌉+1) level require in stack

size of stack (⌈logn⌉+1) k

O(k(logn)) → O(logn)

space require for stack.

```
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)
}
```

* out of place Algo is mergesort.
* Its based on divide & concure.

Time taken to run that code is $O(n \log n)$ mergesort

space $\rightarrow O(n)$

⇒ A time & space for merge $\rightarrow O(n)$

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

By Master theorem

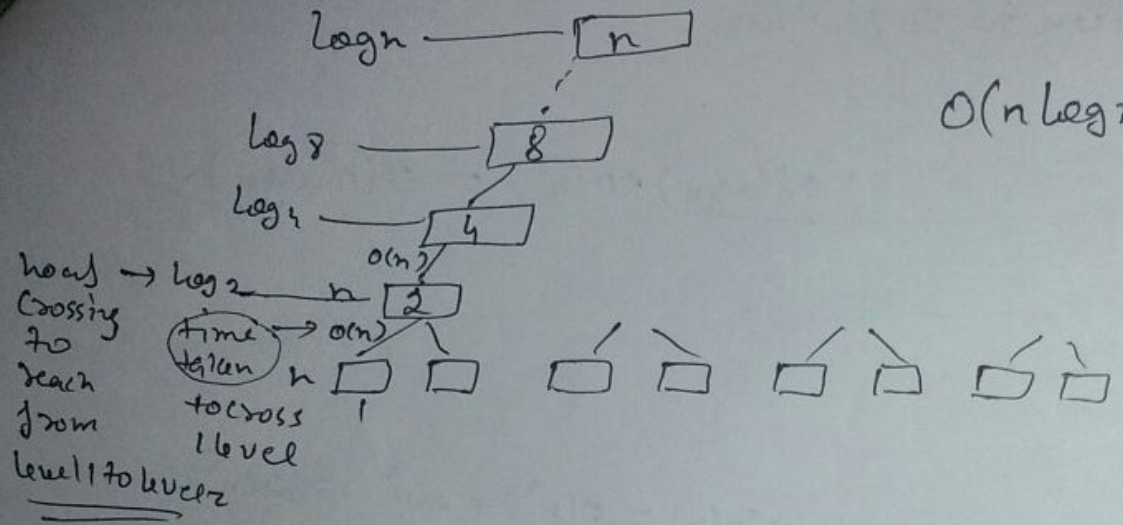
$= O(n \log n)$ ~ for Best & Worst Case

* Two way merging :- 2 list will merge.

~~Two list merge~~

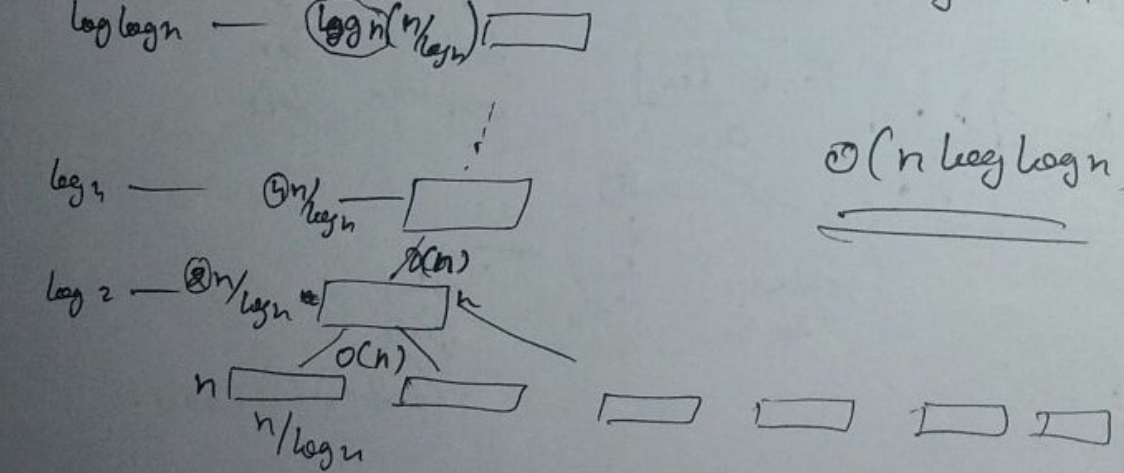
* 2 way & 3 way merging both have diff. time complexity.
* In 3 way merging we merge 3 (elements) list.

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



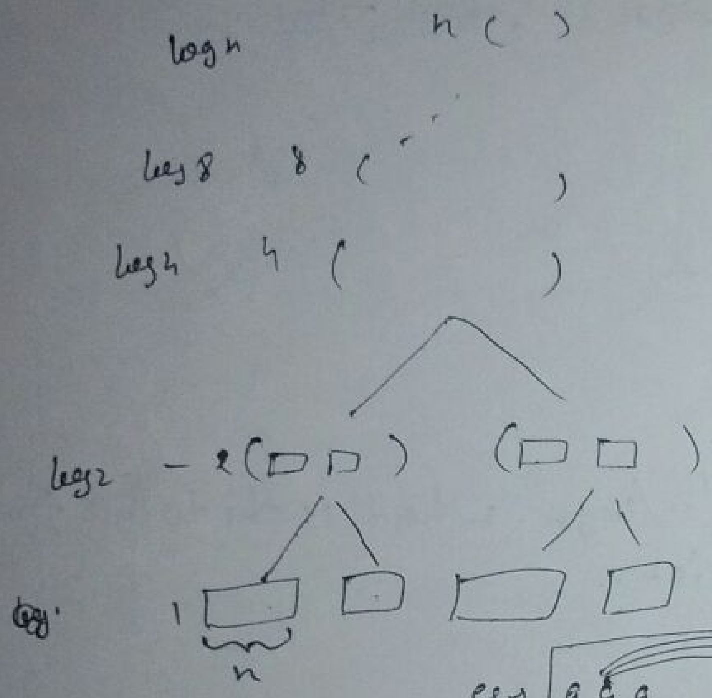
$O(n \log n)$ → Total time.

Q. Given "log n" sorted lists each of size "n/log n" what is the total time required to merge them into one single list.



$O(n \log \log n)$.

Q. 'n' strings each of length 'n' are given then what is the time taken to sort them.



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

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

— $O(n) \times O(n) = O(n^2) + O(n^2) = O(n^2)$ ✓

$O(n^2)$ ↑ for comparing; $O(n^2)$ ↓ for copy

eg→ a a a b a a

aaa baa

O(n)

n … n → O(n) ↓ 1 string take time

(n) ↓ total string

$= O(n^2)$ → total time for comparing of 1 level.

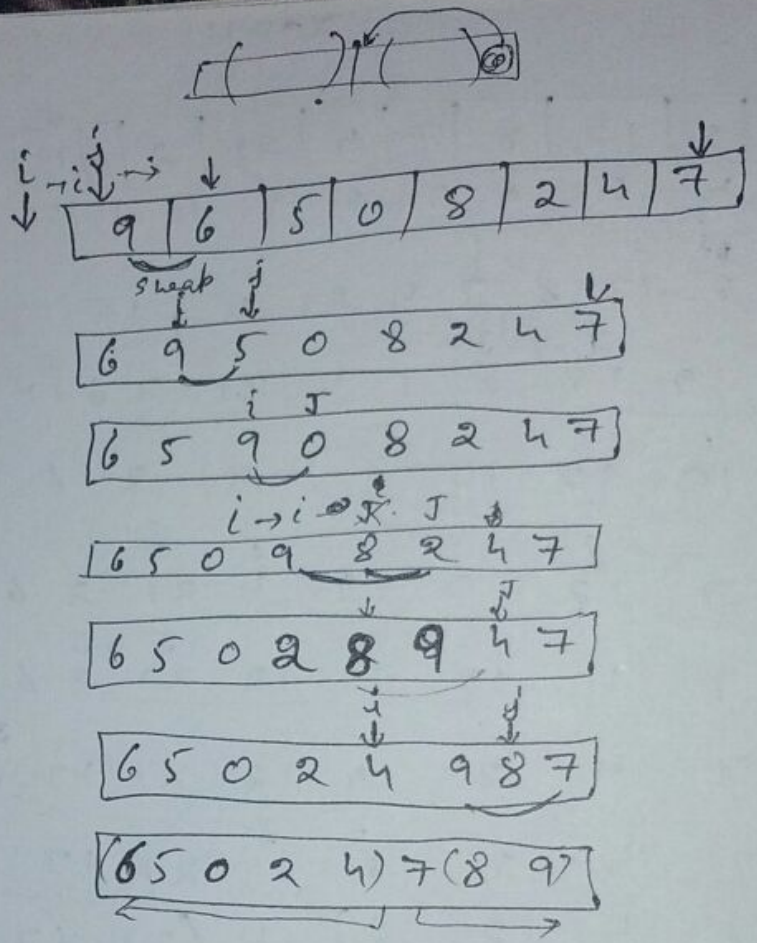
Q.1) Merge sort use as Divide & conquer.

~~Q.2) 'm' & 'n' → O(m+n) → total time~~

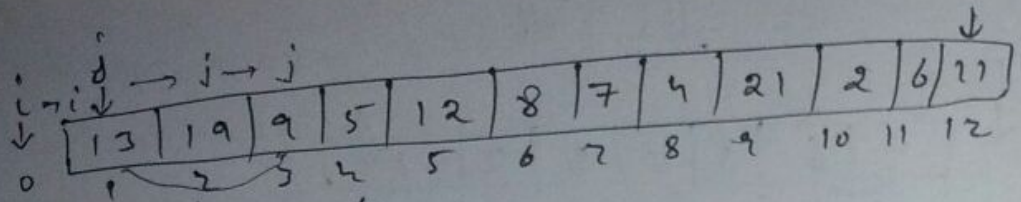
Quick SORT Algo

Partition algo:-

```
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
}
```



x = 11



9 19 13 5 12 8 7 4 21 2 6 11

9 5 13 19 12 8 7 4 21 2 6 11

9 5 8 19 12 13 7 4 21 2 6 11

9 5 8 7 12 13 19 4 21 2 6 11

9 5 8 7 4 13 19 12 21 2 6 11

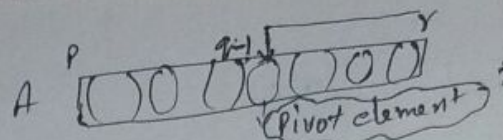
9 5 8 7 4 2 19 12 21 13 6 11

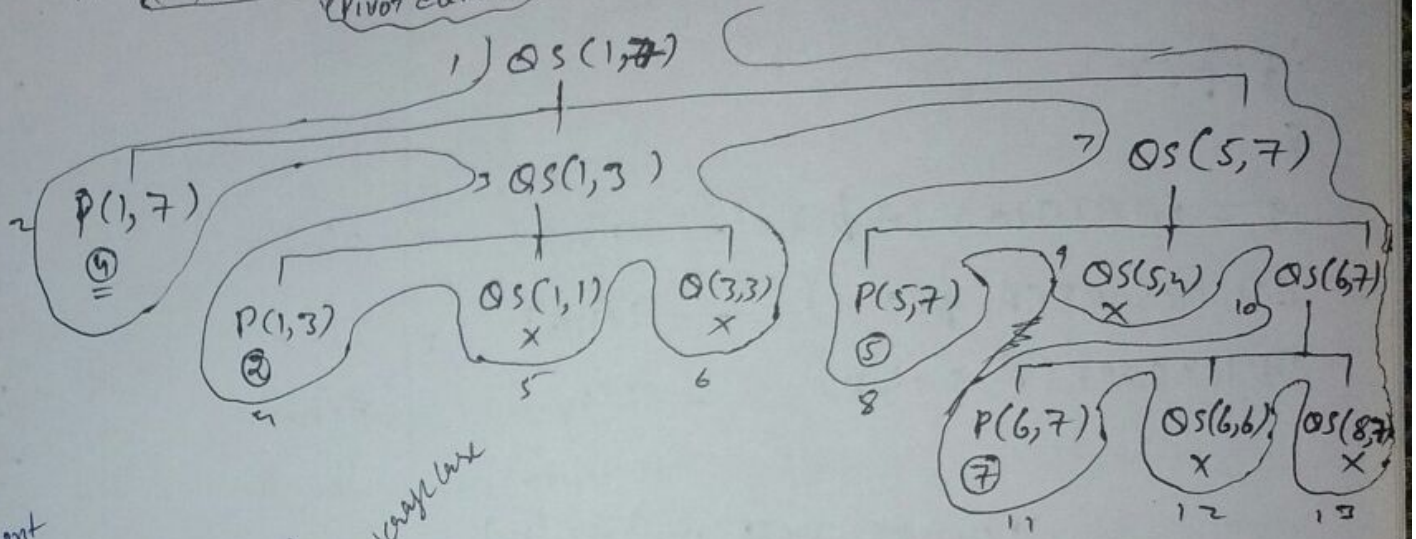
9 5 8 7 4 2 6 12 21 13 19 11

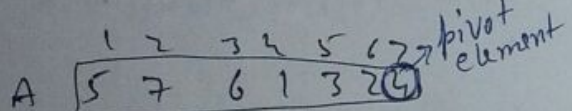
(9 5 8 7 4 2 6) 11 (21 13 19 12)

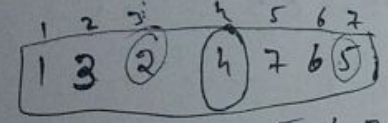
QUICK SORT algo

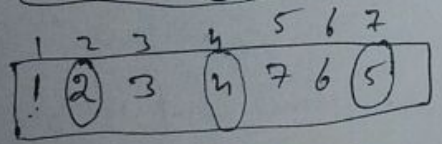
```
QUICKSORT(A,p,r)
{
if(p<r)
{
q = PARTITION(A,p,r)
QUICKSORT(A,p,q-1)
QUICKSORT(A,q+1,r)
}
}
```

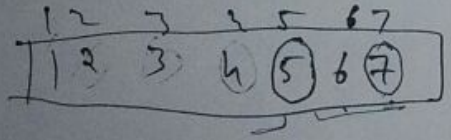




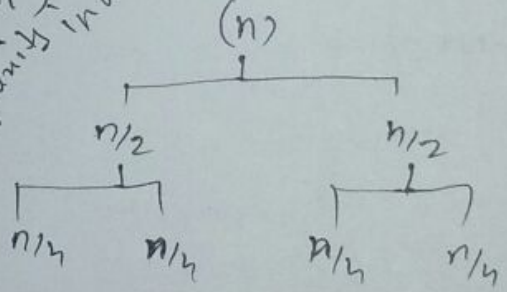


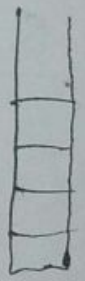






O(log n) ↓ space complexity in average case





In worst case O(n) space complexity

```
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)
```

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

By Masters theorem

Time complexity → $O(n \log n)$

In Best Case.

1	5	6	7	10

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

By Back substitution

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

$T(n-3) + c(n-2) + c(n-1) + cn$

⋮

$c + c \cdot 2 + c \cdot 3 \; - - \; c \cdot 3$

$O(n^2)$

↳ in worst case time complexity.

if element in Asending order

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

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

By Back Sub.

$= O(n^2)$ ⟹ Time complexity

if input in desending order

(6 5 4 3 2 1)

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

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

$= O(n^2)$ ⟹ Time complexity

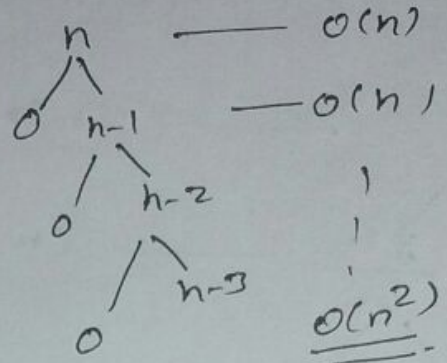
for same element

((2 2 2 2 2) 2)

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

$= O(n^2)$ Time complexity

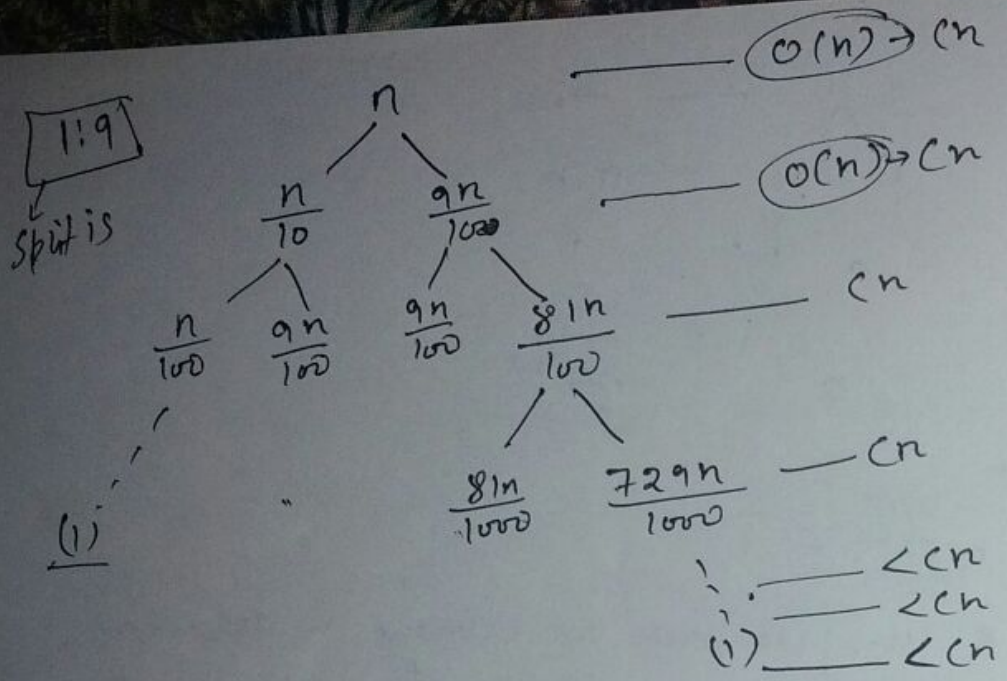
Worst case ⟹ Best case $\Omega(n \log n)$



NOTE! -

* When we have element in assending, desending order or elements are same we have complexity will $O(n^2)$

* When we have split at (0, n-1) or we have once best or once bad split complexity will $O(n \log n)$.

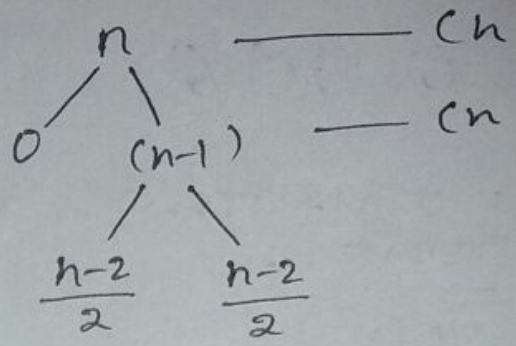


$n - \frac{n}{10/9} - \frac{n}{(10/9)^2}$ ---1

$\log_{10/9} n = O(\log_2 n)$

$O(n \log n)$

* This is when we have split is (1:9) or (0, n-1)
* we split at like 1:99, 1:999, 1:3, 1:1 complexity will $O(n \log n)$.



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

$= 2cn + 2T(n-2)$ → (n) at last.

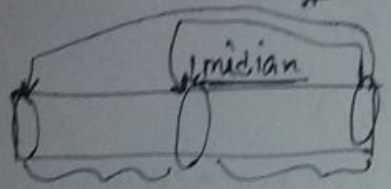
$\leq O(n) + 2T(n/2)$

By master theorem.

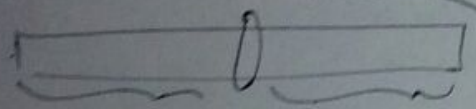
$O(n \log n)$

* This is when we get once bad & once best split.

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 ?



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



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

By master theorem

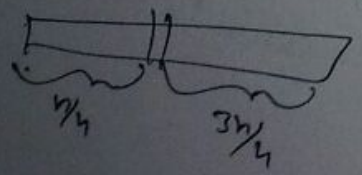
$O(n \log n)$

Q. In 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 ?

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

$T(n) = O(n) + T(n/4) + T(3n/4)$

1 : 3



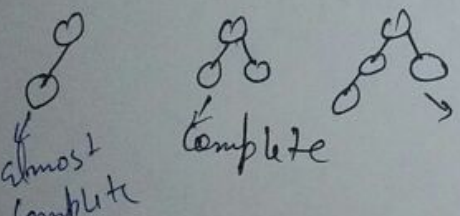
$O(n \log_4 n)$

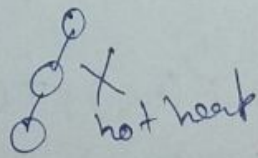
* Heaps

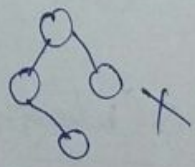
	insert	search	find min	Delete min
unsorted array	$O(1)$	$O(n)$	$O(n)$	$O(n)$
(non decreasing) sorted array	$O(n)$	$O(\log n)$	$O(1)$	$O(n)$
(unsorted) linked list	$O(1)$	$O(n)$	$O(n)$	$O(n+1)=O(n)$
min heap	$O(\log n)$		$O(1)$	$O(\log n)$

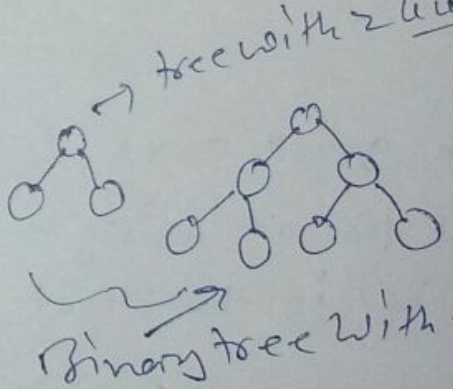
Heap is a binary tree or 3-ary ---- nary tree.

Heap is an almost complete binary tree.

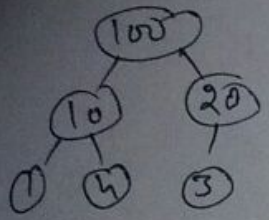




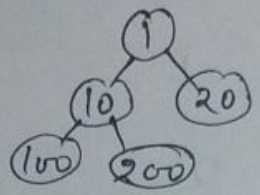




max heap



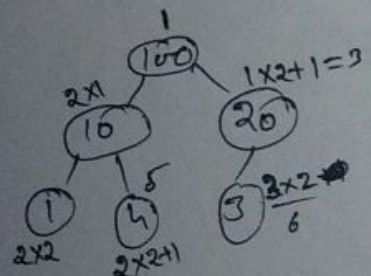
Min Heap



* Multiply by 2 is left shift

Max Heap

* index of parent × 2 → left child
* index of parent × 2 + 1 → right child



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

→ store a binary tree in array

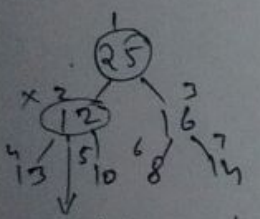
P(i) = ⌊i/2⌋
L(i) = 2×i
R(i) = 2×i+1

* Parent of i
→ left child of i
→ right child of i

⌊5/2⌋ = 2

* ⇒ To find parent of child eg ⌈6/2⌉ = 3

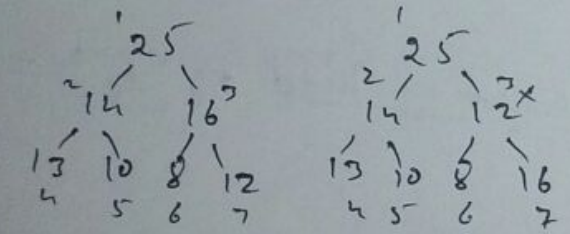
	A.length	A.heapsize → How element follow heap
25, 12, 16, 13, 10, 8, 14	7	1
	7	7
25, 14, 16, 13, 10, 8, 12	7	1
25, 14, 13, 16, 10, 8, 12	7	2
25, 14, 12, 13, 10, 8, 16	5	5
14, 13, 12, 10, 8	5	5
14, 12, 13, 8, 10	5	5
14, 13, 8, 12, 10	5	5
14, 13, 12, 8, 10	13	2
89, 19, 40, 17, 12, 10, 25, 7, 11, 6, 9, 70		

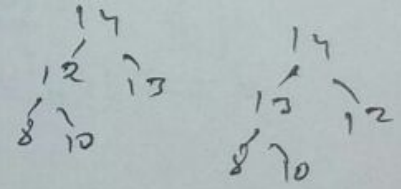


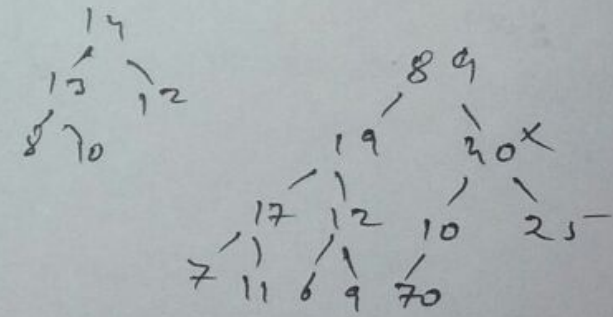
max. is not present at root so it not follow heap property

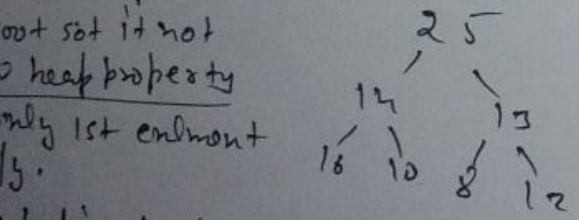
So only 1st element satisfy.

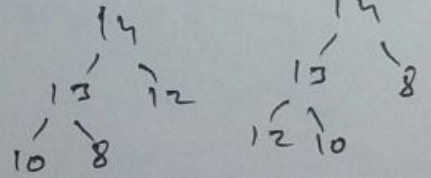
* every leaf is a heap











* if array is in ascending order then it is min heap [8, 10, 12, 13, 14]

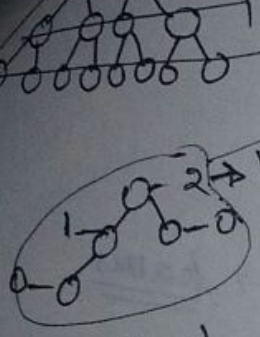
* if array is in decending order it is max heap [14, 13, 12, 10, 8]

→ H(T) = 3

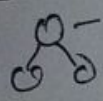
H(T)	1	2	3	4
max node	3	7	15	31

*

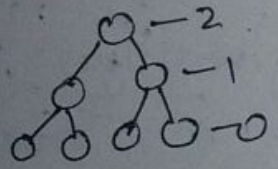
$2^{h+1} - 1$ Here h = height of tree



H(T) = 2 *This rule not apply on this tree.



Height of tree = 1



H(T) = 2

node of complete binary tree or almost complete binary tree

Height of tree

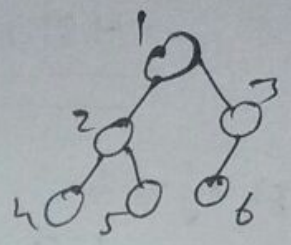
$n \rightarrow \lfloor \log n \rfloor$

3 → 1
7 → 2
15 → 3

$n - \lfloor \log_2 4 \rfloor = (2)$

* { $O(\log n)$ → Height of binary tree almost heap.

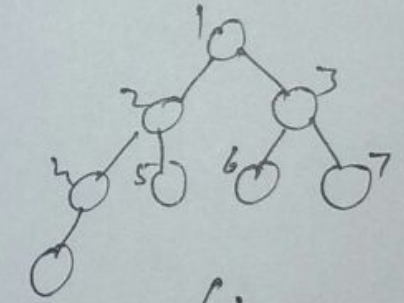
```
MAX-HEAPIFY (A,i)
{
  l = 2i;
  r = 2i+1;
 if (l ≤ A.heap-size and A[l] > A[i])
   largest = l;
else largest = i;
if (r ≤ A.heap-size and A[r] > A[largest]
   largest = r;
if (largest ≠ i)
exchange A[i] with A[largest]
  MAX-HEAPIFY (A, largest)
}
```



$(\lfloor \frac{n}{2} \rfloor + 1$ ton $) \rightarrow$ leaves

$(\lfloor \frac{6}{2} \rfloor + 1$ to 6 $)$

(4 to 6) → it gives vertices in a tree.



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

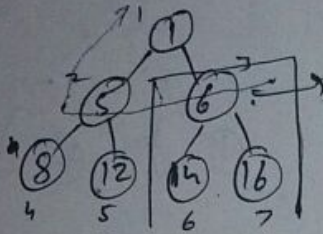
$(\lfloor \frac{8}{2} \rfloor + 1$ to 8 $)$

(5 to 8) → leafs

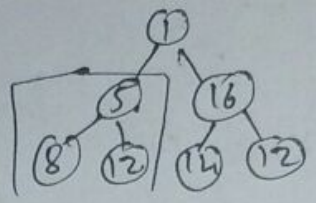
1

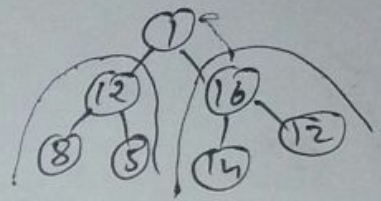
* A leaf is already a heap.

1 5 6 8 12 14 16



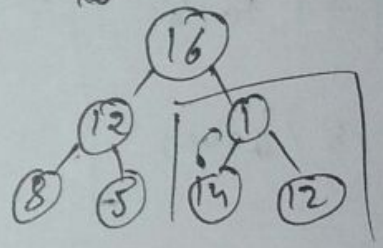
Compare root with L & R leaf

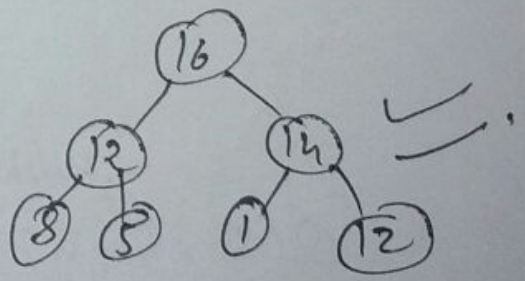




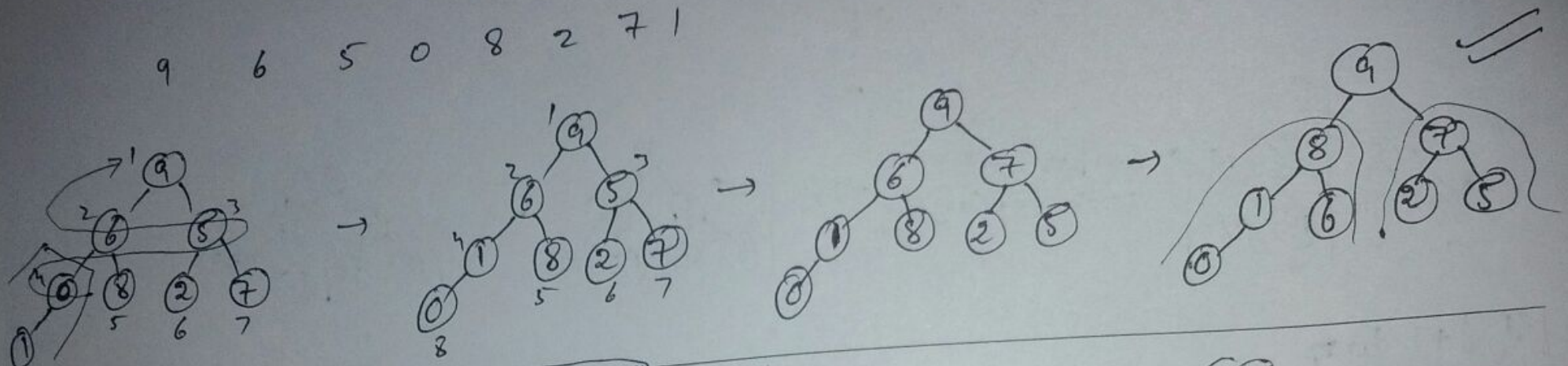
$\lfloor \frac{7}{2} \rfloor + 1$ to 7

(4 to 7) = no of leafs





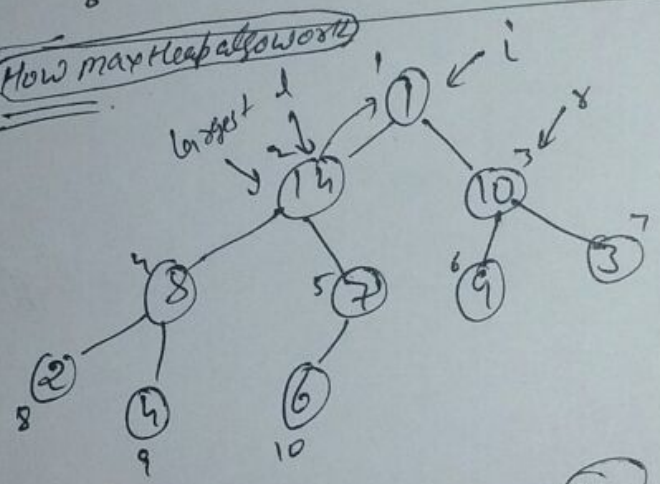
9 6 5 0 8 2 7 1

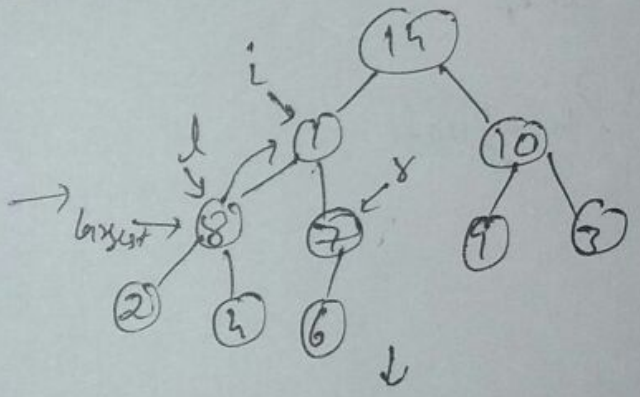


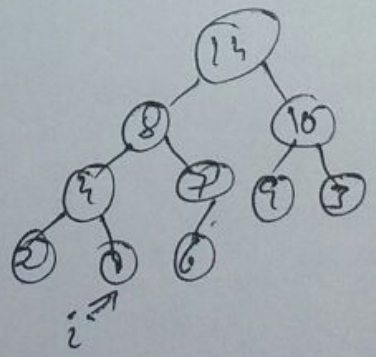
$\lfloor \frac{8}{2} \rfloor + 1$ to 8

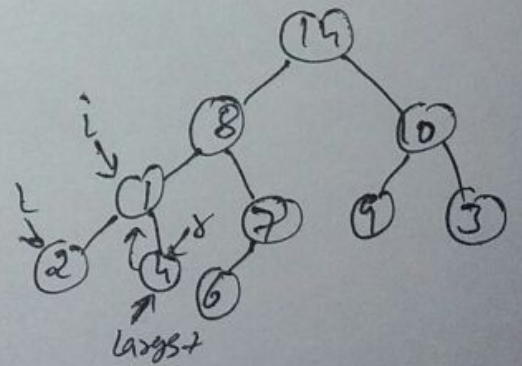
(5 to 8) leaves

How maxHeapalgoworks









Heapify

$O(2 \times \log n)$

Time → $O(\log n)$

Time complexity in worst case

$O(\log n)$ → space complexity

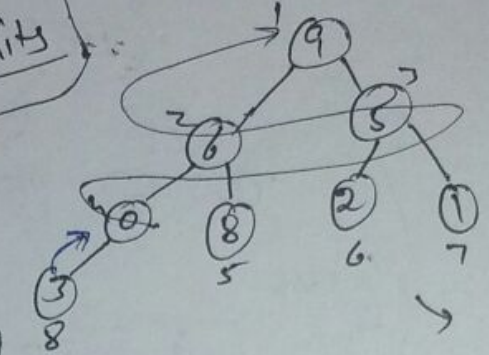
```
BUILD-MAX-HEAP(A)
{
A.heap-size = A.length
for (i = ⌊A.length/2⌋ down to 1)
  MAX-HEAPIFY (A,i)
}
```

O(log n) — space complexity

9 6 5 0 8 2 1 3

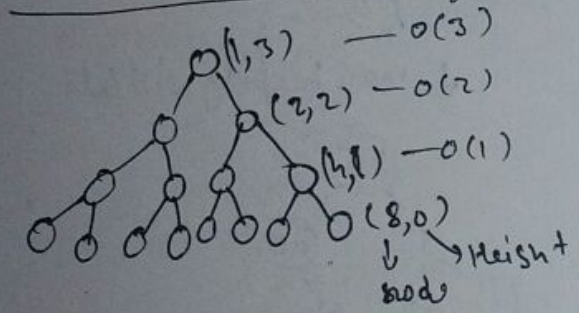
$\lfloor 8/2 \rfloor + 1$ to 8

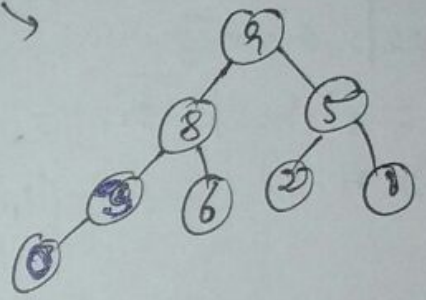
(5 to 8)



(log n) ↓ Height of tree

time taken





$\sum_{h=0}^{\log n} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h)$

$O(n \log n)$ — time taken

$\left\lceil \frac{n}{2^{h+1}} \right\rceil \rightarrow \left\lceil \frac{15}{2^{0+1}} \right\rceil = 8$ → no of nodes of height 0.

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

$\left\lceil \frac{15}{2^{1+1}} \right\rceil = 4 \qquad \left\lceil \frac{15}{2^{3+1}} \right\rceil = 1$

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

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

time taken

space → O(log n)

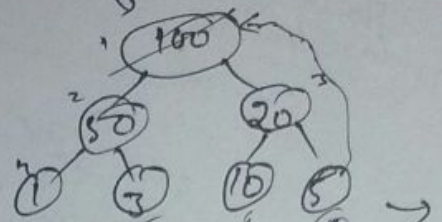
* Delete maximum in a heap

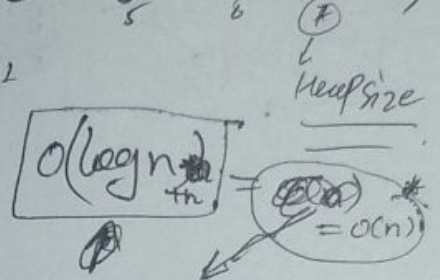
```
Heap-Extract-max(A)
{
if A.heap-size<1
error "heap underflow"
max = A[1]
A[1] = A[A.heap-size]
A.heap-size = A.heap-size-1
max-HEAPIFY(A,1)   — O(log n)
return max
}
```

} constant time taken

100 50 20 1 3 10 5

delete 100 then



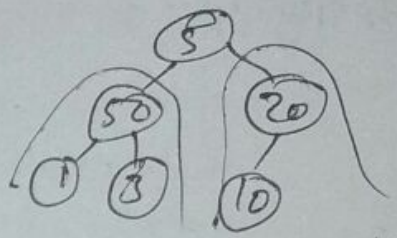


HeapSize

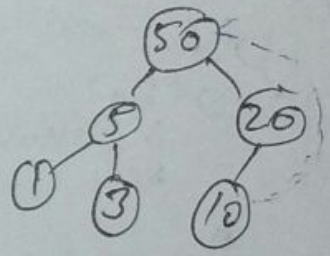
Total time taken

space complexity
= O(log n)

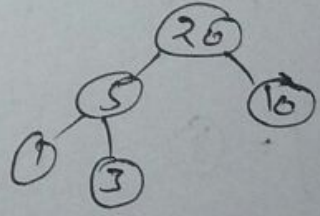
140

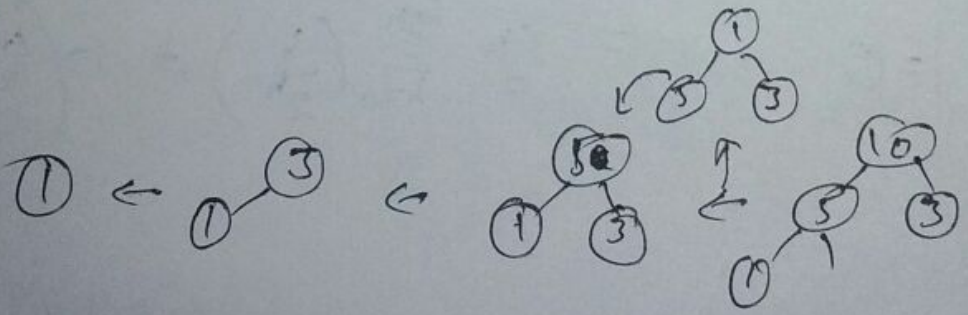


↓ max heapify will call



Again delet max. then





Incr. value of any node

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

it may not go beyond to root

parent

O(1) ↑

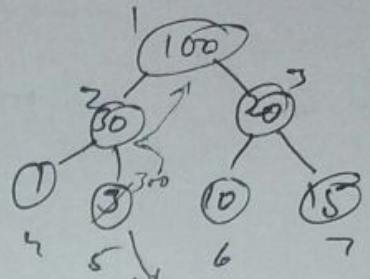
space complexity

O(logn)

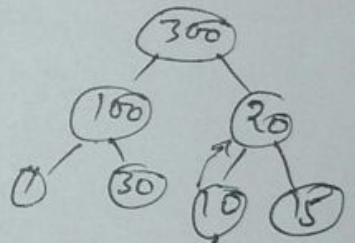
↑ time in incr. key

O(logn)

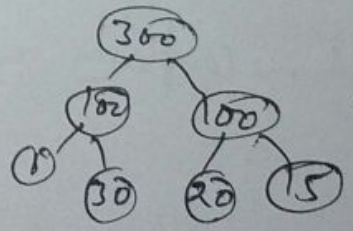
time in decr. key



if we want to incr. 3 to 300. then

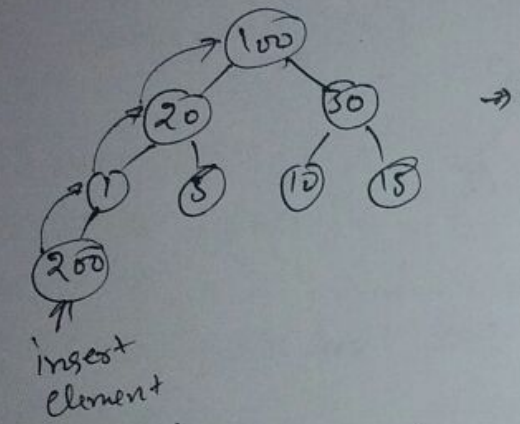


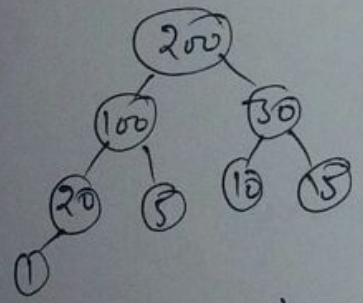
incr. by 100 then



=

: How insert element in max heap





O(logn) time complexity

	findmax	deletemax	insert	incr.	decr.
				O(logn)	O(logn)
max heap	O(1)	O(n+logn)=O(n)	O(logn)		

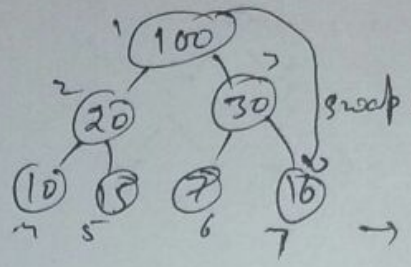
Find min	search	delete
O(n)	O(n)	O(n+logn)= logn)

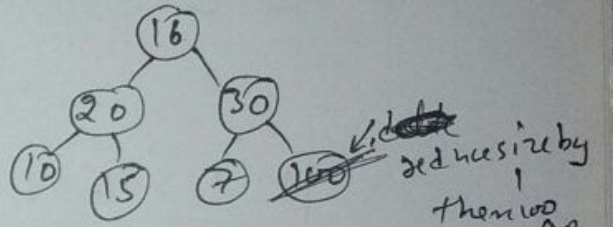
Heap sort & analysis

O(n log n) → time complexity

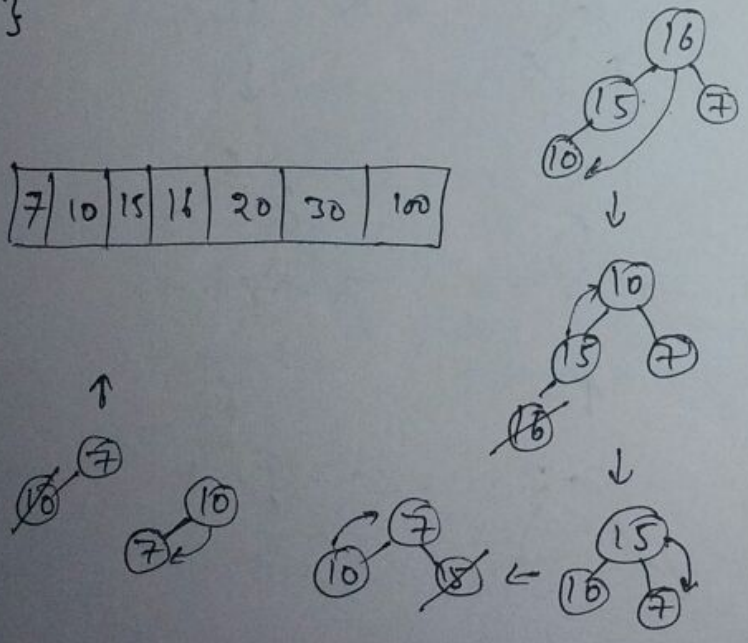
100	20	30	10	15	7	16

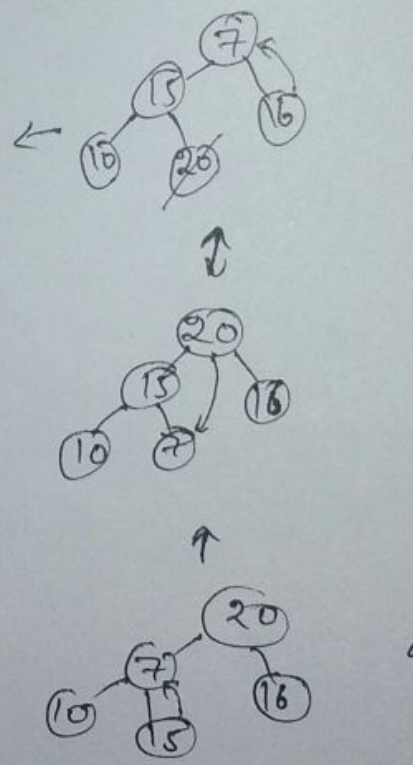
```
Heap(A)
{
BUILD-MAX-HEAP(A)
for (i = A.length down to 2)
 exchange A[i] with A[i]
 A.heap-size = A.heap-size-1
 MAX-HEAPIFY (A, 1)
}
```

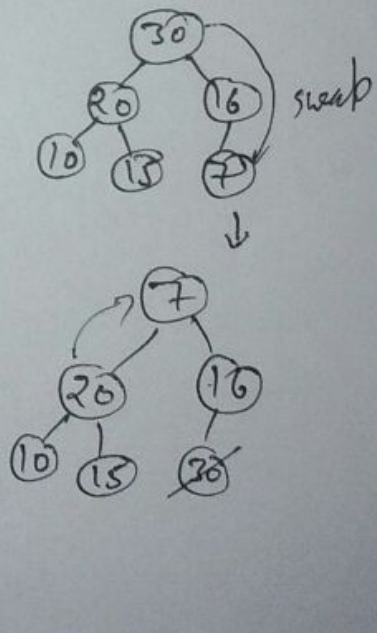


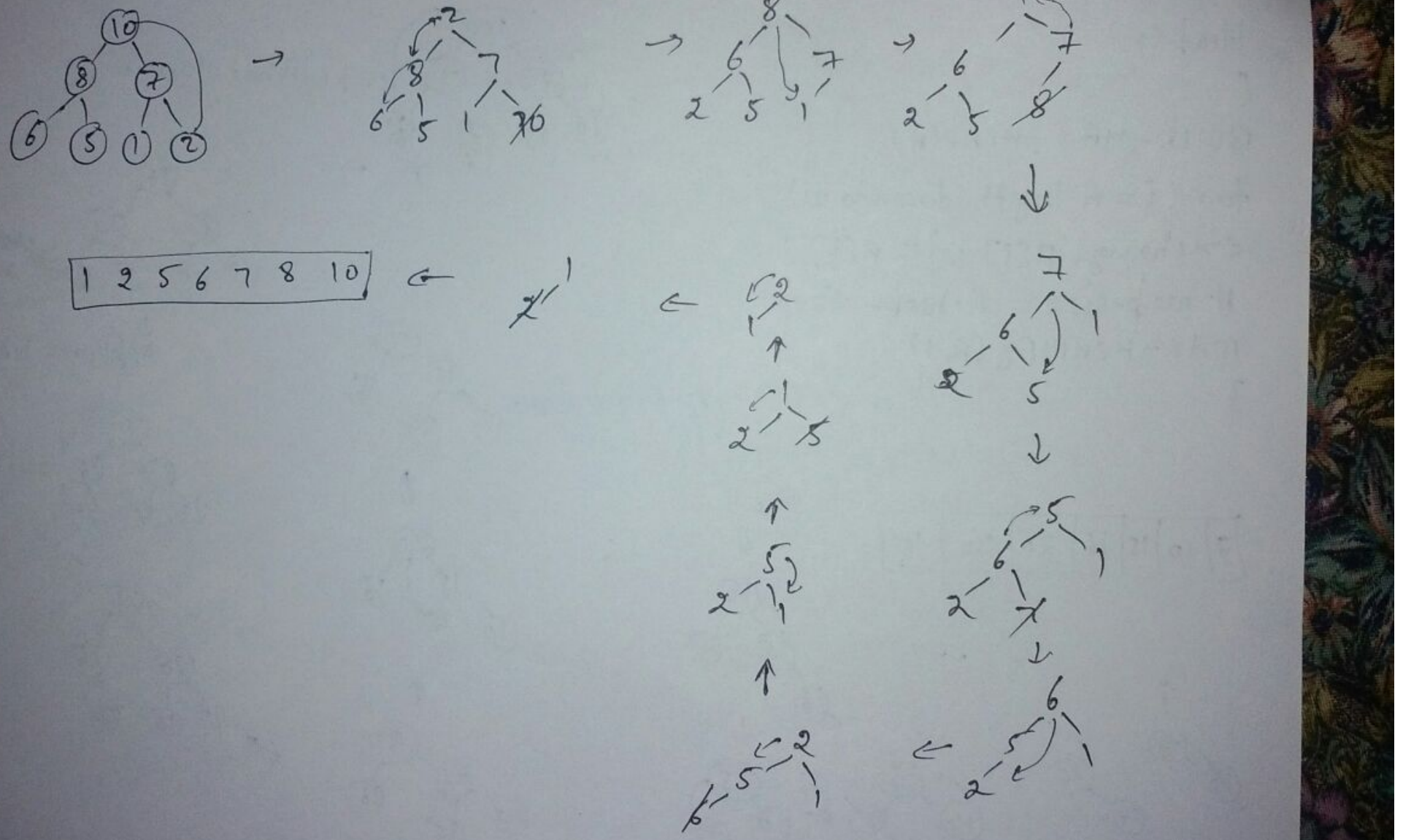


7	10	15	16	20	30	100








10 8 7 6 5 1 2
1 2 5 6 7 8 10

Problems on Heap & Heap sort

In a heap with n elements with the smallest element at the root, the 7th smallest element can be found in time

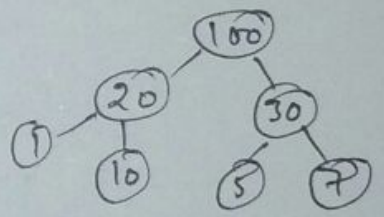
a) $O(n \log n)$ b) $O(n)$ c) $O(\log n)$ d) $O(1)$

delete min – $O(\log n)$
delete 2 min – $O(\log n)$
3 – $O(\log n)$
4 – " "
5 – " "
6 – " "
Find 7 min – ~~$O(\log n)$~~ $O(1)$
Insert – 6 * $O(\log n)$
all element which delet $= O(\log n)$

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

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

Let max heap like



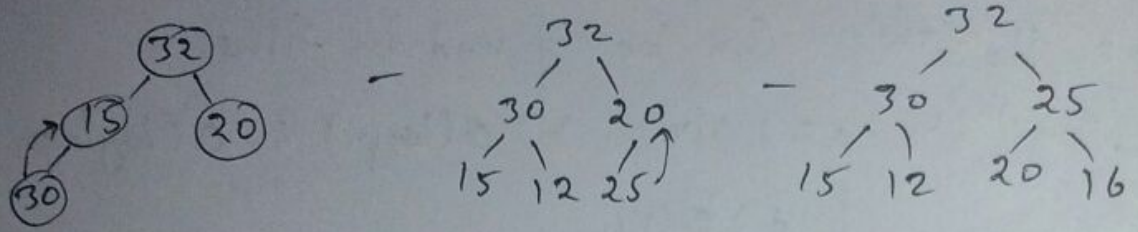
$\lfloor n/2 \rfloor + 1$ ton

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

Q.

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

max heap



32	30	25	15	12	20	16

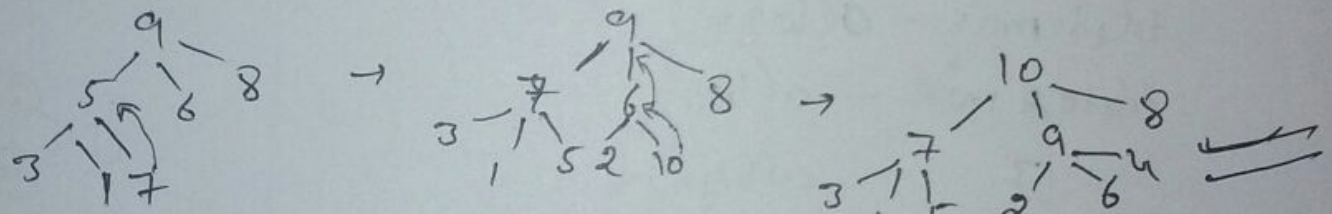
Q. using Ternary which gives max heap

1, 3, 5, 6, 8, 9

9, 6, 3, 1, 8, 5

9, 3, 6, 8, 5, 1

9, 5, 6, 8, 3, 1



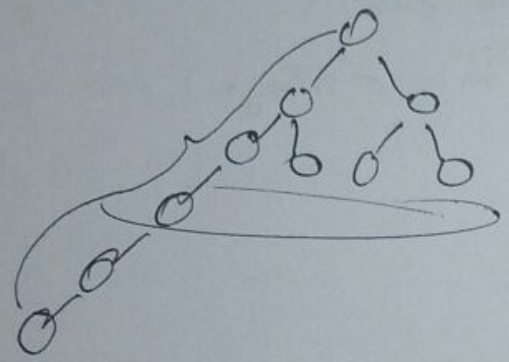
then insert

7, 2, 10, 4

10	7	9	8	3	1	5	2	6	4

array.

Q. 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 number of comparison performed are.



O(logn) → height of tree

O(loglogn)

Q. we have a binary heap on 'n' elements & wish to insert 'n' more elements (not necessarily one after another) in to this heap. The total time required for this is

a) O(logn) ~~b) O(n)~~

c) O(nlogn) d) O(n²)

→ Use Build heap & {Suppose there are not n element at first}.

O(2n) = O(n)

Q. GATE-16

Assume that the algorithms considered here sort the input sequences in ascending order. If the input is already in ascending order, which of the following is true?

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

II. Bubble sort runs in $O(n^2)$ time. → it take $O(n)$

III. Merge sort runs in $O(n)$ time. — it take $O(n \log n)$

IV. Insertion sort runs in $O(n)$ time.

a) I & IV only b) I & III only

c) II & IV only d) I & IV only

GATE-16 The worst case running time of Insertion Sort, Merge Sort & Quick Sort, respectively are:

a) $O(n \log n)$, $O(n \log n)$ & $O(n^2)$

b) $O(n^2)$, $O(n^2)$ & $O(n \log n)$

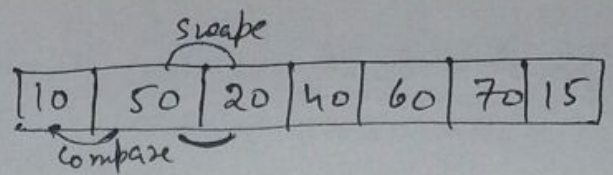
c) $O(n^2)$, $O(n \log n)$ & $O(n \log n)$

d) $O(n^2)$, $O(n \log n)$ & $O(n^2)$

Bubble sort algorithm

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

→ It will check in earlier iteration is swapping occurs or not if not that means element are sorted.



10 20 50 40 60 70 15

10 20 40 50 60 70 15

10 20 40 50 60 15 | 70

10 20 40 50 15 | 60 70

10 20 40 15 | 50 60 70

10 20 15 | 40 50 60 70

10 15 20 40 50 60 70 ✓

---

* eg →

10 20 30 40 50 → only one iteration occur after that flag will 1 (swapped = 1)

---

*

10 20 30 40 60 50 swapped = 1

10 20 30 40 50 | 60 swapped = 0

then it means element are sorted

*Time complexity of Bubble sort

first comparision

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

$\frac{(n-1)(n-2)}{2} = O(n^2) \rightarrow$ Worst case.

$(n-1) = O(n) \rightarrow$ Best case.

or

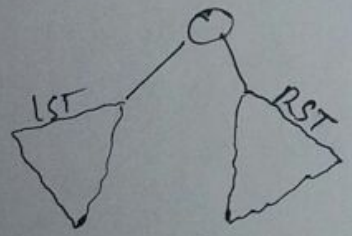
if two comparision

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

...t-on B.Sort
GATE-16

Consider a complete binary tree where the left & right subtrees of the root are max-heaps. The lower bound for the number of operations to convert the tree to a heap is -

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



zero — left subtree

$O - \log n$

$O(\log n)$

GATE15

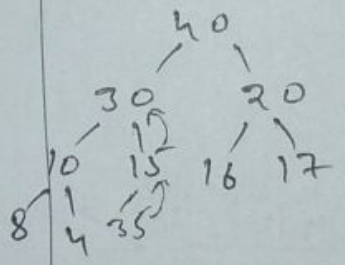
Consider a max-heap, represent by the array 40, 30, 20, 10, 15, 16, 17, 8, 4. Now consider that a value 35 is inserted into this heap. After insertion the new heap is.

a) 40, 30, 20, 10, 15, 16, 17, 8, 4, 35

b) 40, 35, 20, 10, 30, 16, 17, 8, 4, 15

c) 40, 30, 20, 10, 35, 16, 17, 8, 4, 15

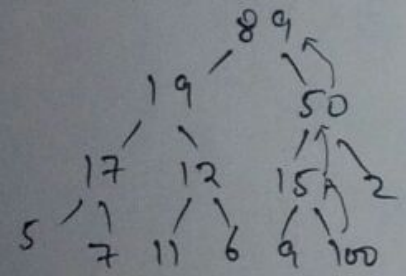
d) 40, 35, 20, 10, 15, 16, 17, 8, 4, 30



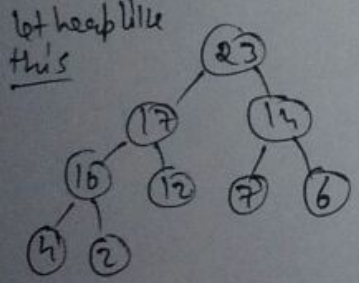
GATE-15

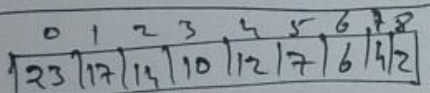
Consider the following array of elements
<89, 19, 50, 17, 12, 15, 2, 5, 7, 11, 6, 9, 100>
The mini. no of interchanges needed to convert it into a max heap is

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



let heap like this





GATE-14

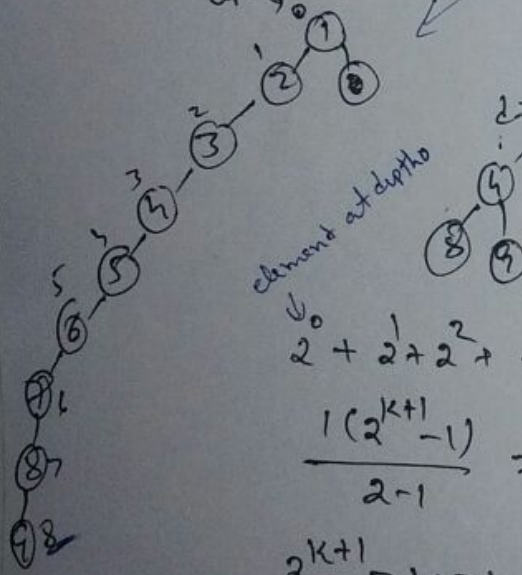
Q.) A operator delete(i) for a binary heap DS is to be designed to delete the item in the ith node. Assume that the heap is implemented in an array and i refers to the ith index of the array. If the heap tree is depth 'd' (no of edges on the path from the root to the farthest leaf) then what is the time complexity to refix the heap efficiently after the removal of the element.

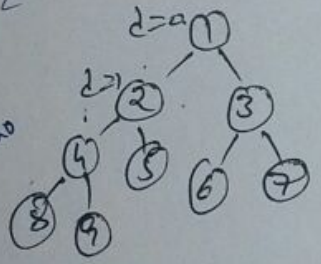
a) O(1) ~~b) O(d) but not O(1)~~

c) $O(2^d)$ but not O(d)

d) $O(d2^d)$ but not $O(2^d)$

A complete binary min heap is made by including each int. 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 maximum depth at which integer '9' can appear is 8 ⇒ Ans





$2^0 + 2^1 + 2^2 + \cdots 2^K = 1023$

$\frac{1(2^{K+1}-1)}{2-1} = 1023$

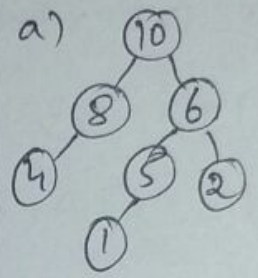
$2^{K+1} = 1024$

$2^{K+1} = 2^{9}$

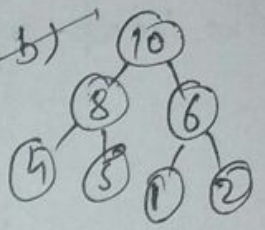
that means its possible this case. $K = 9$

A max heap is a heap where the value of each parent is greater than or equal ~~to~~ to the values of its children. Which of the following is a max heap

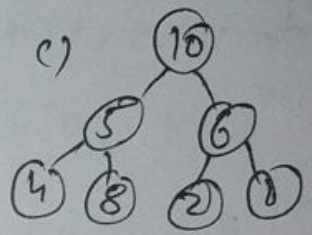
a)



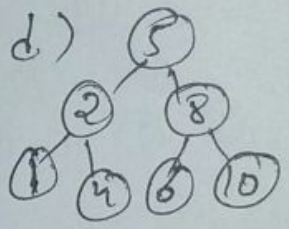
b)



c)



d)

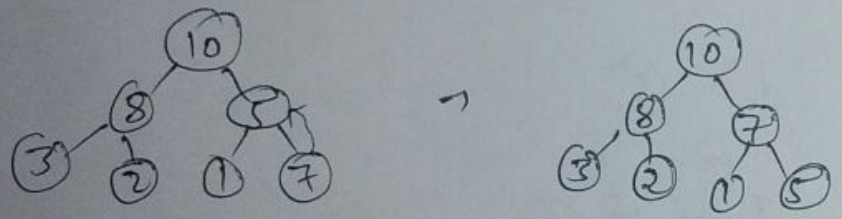


GATE-14

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. Two new elements 1 & 7 are inserted into the heap in that order. The level order traversal of the heap after insertion of the element is

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

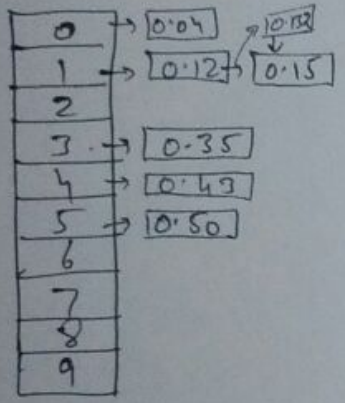


→

Bucket Sort

Sort a large set of floating point no which are in the range 0.0 to 1.0 & are uniformly distributed across the range?

0.35, 0.12, 0.43, 0.15, 0.04, 0.50, 0.132



0.04
0.12
0.132
0.15
0.35
0.43
0.50

~~O(1) Time complexity if uniform.~~

O(n) → Best case time complexity

O(n+K) → space complexity
↓
no of bucket

$O\left(\left(\frac{n}{K}\right)n\right)$

$O(n^2)$ worst case time complexity

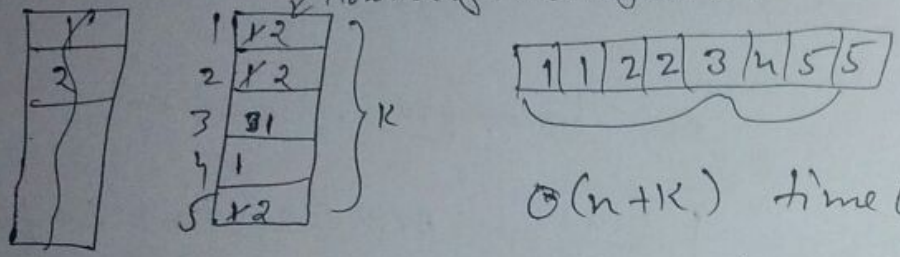
* Counting sort

(1 — 5)

* It can apply only bcoz of no is given

eg (1—5) means no not go beyond 1 & more than 5.

2, 2, 3, 4, 1, 5, 5



$O(n+K)$ time complexity

$O(K)$ space complexity

Radix-sort(aid) → array containing element
Radix-sort(A,d) → no of digit

* Each key in A[1...n] is a d-digit integer.
* Digits 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.

(3 2 1)	sorting	sorting	sorting
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

O(b) space complexity
↓
base.

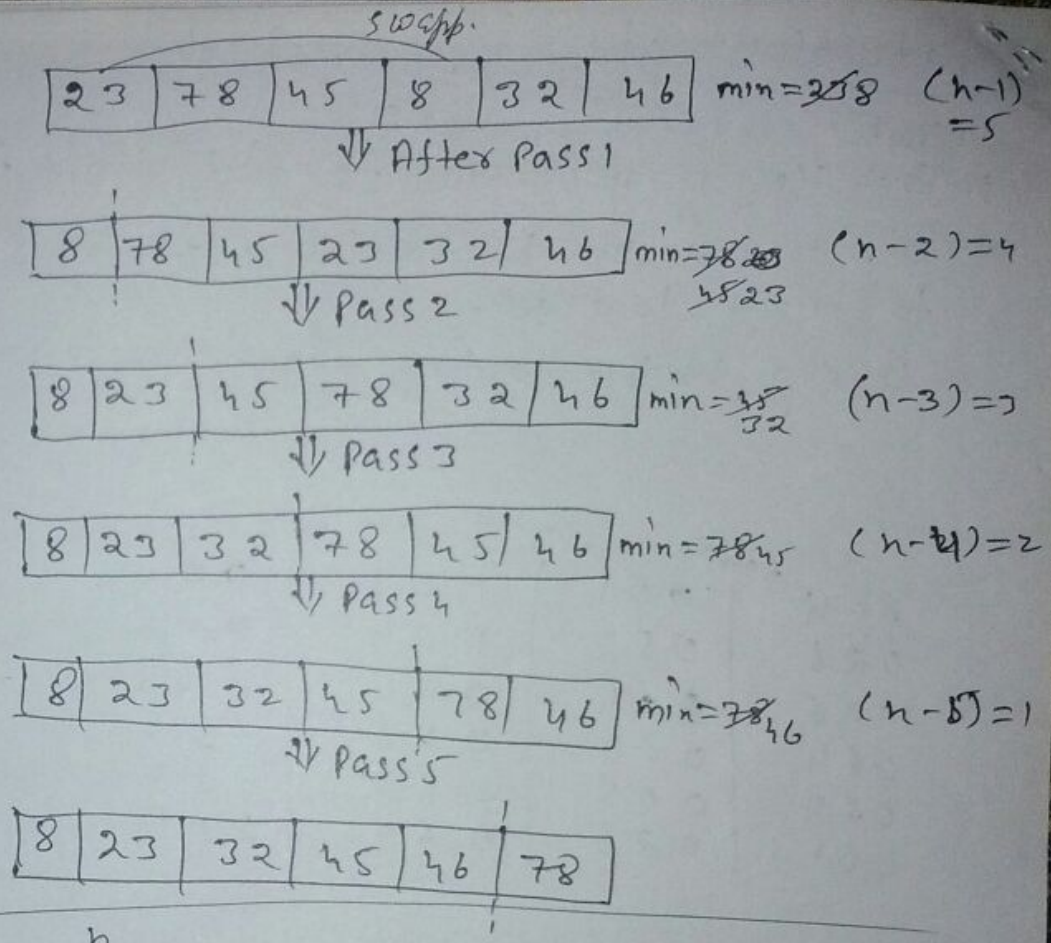
d(o(n+b))
↓ no of digit (can present in largest no.)
↓ size of array
↓ base

let no is d then log_b d (o(n+b)) time complexity.

if d = very big = O(n^k) then
k log_b n (o(n+b))
↓ constant
O(n log_b n)
↓ in worst case.

* Algo. of selection sort file.

```
void selection sort (int a[], int n)
{
int i, j, temp, min;        // temp → used for swapping
for (i=0; i<n-1; i++)       → for pass.
{
 min = i;
 for (j=i+1; j<n; j++)
 if (a[j] < a[min]) min = j;     → used to find min in pass
 temp = a[i];
 a[i] = a[min];                  } swapping logic
 a[min] = temp;
 }
 }
```



n
$n-1$
$n-2$
$n-3$
1

$n-(n-1) = 1$

$\frac{(n-1)n}{2} = O(n^2)$ time complexity.

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

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

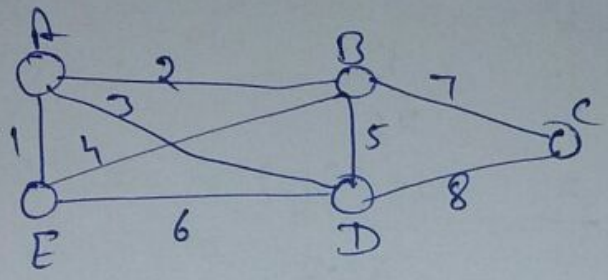
$O(1)$ space complexity because we use only 1 extra variable.

Greedy Algo

1) optimization :- minimizing & maximizing of things.

Ex :- min cost , max profit
min risk , max reliability.

Ex



find shortest path from A to C.

* Now we have one option is to check each possibility from A to C. called Exastive search. which gives complexity of $O(2^n)$ → exponential
* Exponential complexity is not good so this can be solve by Greedy method. which gives $O(n^k)$ → polynomial
* We can apply dynamic problem method but for some values it complexity will $O(n^k)$.
But for some values we get exponential complexity.
So we apply Greedy method in that condition.
Its like applying exastive search in this condition.

Dynamic ← Optimization → Greedy

* Greedy → Not able to solve all problem. (optimization)

* Knapsack (bag) problem

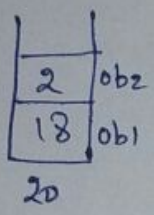
Ex:-

	ob1	ob2	ob3
Profit	25	24	15
weight	18	15	10
P/W	$25/18 = 1.4$	$\frac{24}{15} = 1.6$	$\frac{15}{10} = 1.5$

$n = 3$

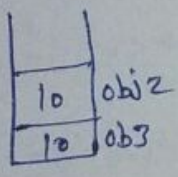
$m = 20$ → capacity of bag.

* Greedy profit



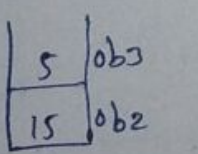
	W	P
ob1	18	25
ob2	2	24 (2/15)
	20	28.2

* Greedy for weight



	W	P
ob3	10	15
ob2	10	24 (10/15)
	20	31

* Greedy for Profit/Weight :- find weight for per unit



	W	P
ob2	15	24
ob3	5	15 ($\frac{5}{10}$)
	20	31.5

Algo for Knapsack

```
Greedy knapsack
{
 for i=1 to n;  — O(n)
 Compute pi/wi;
```

→ for sorting elements apply sorting- Merge sort so its complexity will O(nlogn)

```
Sort object in non increasing (decreasing) order of p/w;  — O(nlogn)

for i=1 to n from sorted list
if (m>0 && wi≤m)
   m=m-wi;
   p= p+pi;        } — O(n)
 else break;
if (m>0)
 p=p+pi(m/wi);  — O(1)
}
```

complexity = O(nlogn)

Ex :-

object	1	2	3	4	5	6	7
Profit	10	5	15	7	6	18	3
weight	2	3	5	7	1	4	1
P/W	5	1.6	3	1	6	4.5	3

$m = 15$, $n = 7$

Sorted order (obj) →

5	1	6	3	7	2	4

fract ←
2
7
3
6
1
5

$m = 15\ 14\ 12\ 8\ 5\ 2\ 0$

$P = 6 + 10 + 18 + 15 + 3 + 5(2/3)$

$= 55.3$

Ex :-

obj.	1	2	3	4	5
P	2	28	25	18	9
W	1	4	5	3	3
P/W	2	7	5	6	3

$m = 15$, $n = 5$

Sorted array of obj. →

2	4	3	5	1

5
3
4
2

$m = 18\ 11\ 8\ 5\ 0$

$P = 28 + 18 + 25 + 9$

E!-

If all the object. weight is same then no need to find P/W. Solve on the behalf of these profits.

oP	2	28	25	18	9
W	4	4	4	4	4

Huffman Coding

Algo

```
Huffman(c)
{
  n=|c|
  make a min heap 'O' with c   - O(n)

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

* Time complexity = O(n log n)
* space complexity = O(n) [∵ depth of tree]

c = set of all character given
z = new node

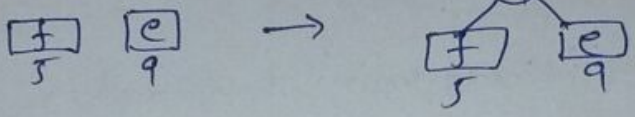
Ex:-

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

Step1 Build a min Heap & Select mini. element

45	13	12	16	9	5

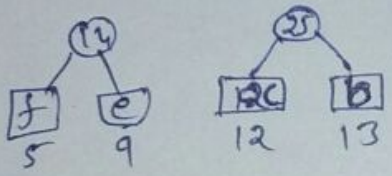
2 Select next mini. element & add them



Step2

45	13	12	16	14

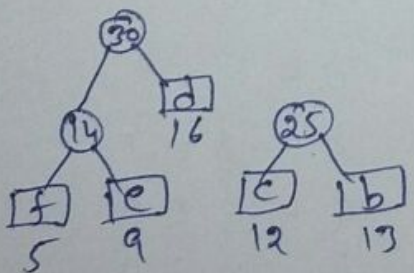
(i) select mini. & (ii) next mini. & add them



Step3

45	25	16	14

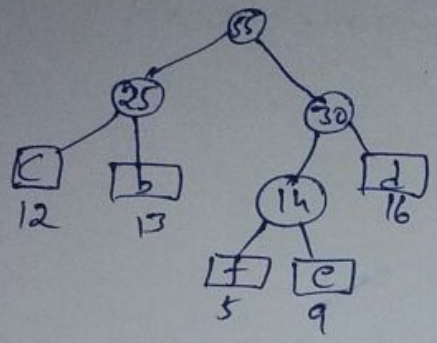
(i) select mini (ii) next mini & add them



Step-4

45	25	30

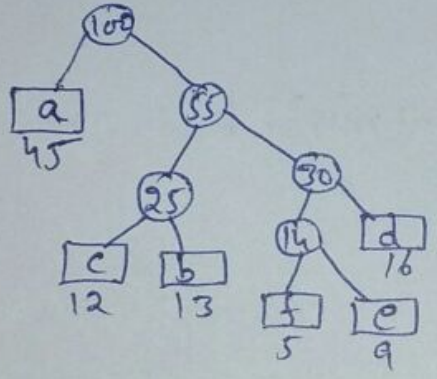
(i) select mini (ii) next mini & add them



Step-5

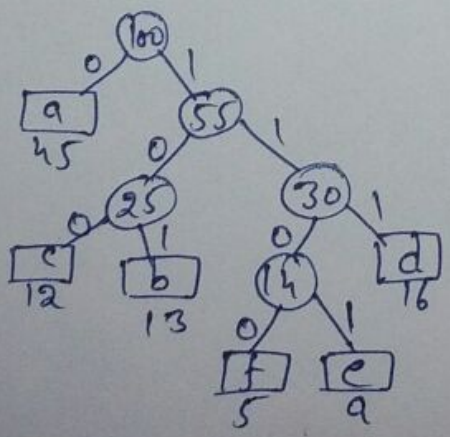
45	55

(i) select mini (ii) next mini & add them



★ Now ~~ther are two way of~~

(i) Now apply a pattern [:' gives smaller or left side either 0 or 1] Here we use 0 to left & 1 for right side.



a - 0
b - 1 0 1
c - 1 0 0
d - 1 1 1
e - 1 1 0 1
f - 1 1 0 0

(ii) Now count the bit used for each character

a - 0 → 1
b - 101 → 3
c - 100 → 3
d - 111 → 3
e → 1101 → 4
f → 1100 → 4

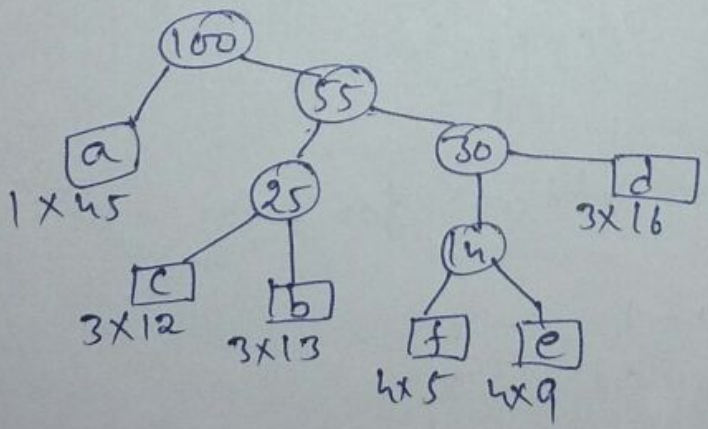
(iii) multiply bits with given value of char & add them

45(1) + 13(3) + 12(3) + 16(3) + 9(4) + 5(4)

45 + 39 + 36 + 48 + 36 + 20 =

SHORTCUT

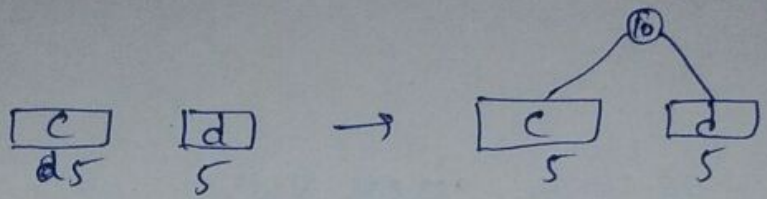
In tree count how many steps require to reach that node & multiply that no with given value of that character. & add all them.



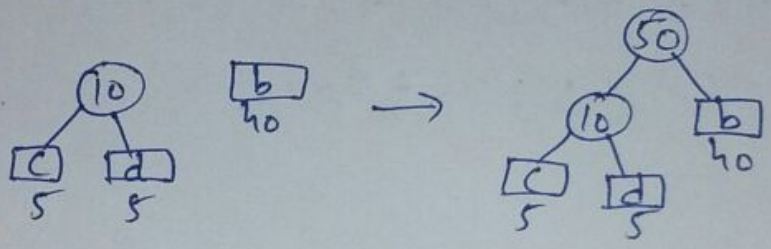
Ex 2

a - 50
b - 40
c - 5
d - 5

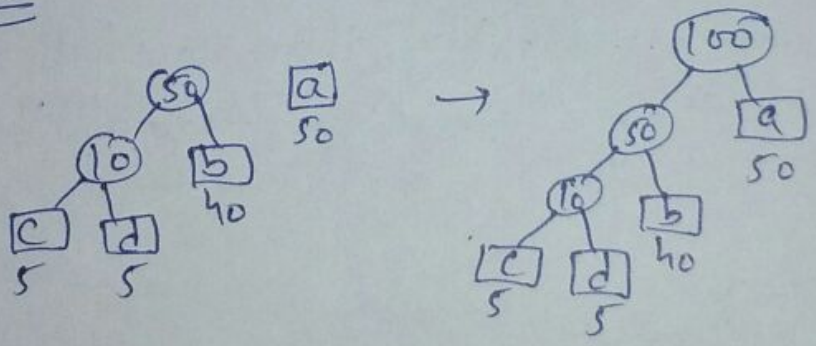
Step-1



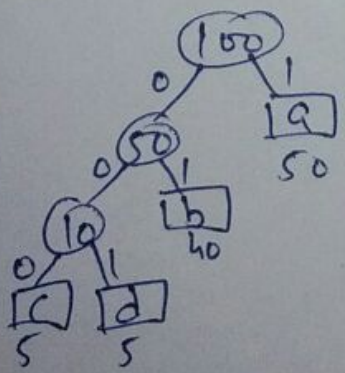
Step-2



Step-3



Step-4 use either 0 for left or 1 for right & vice versa.



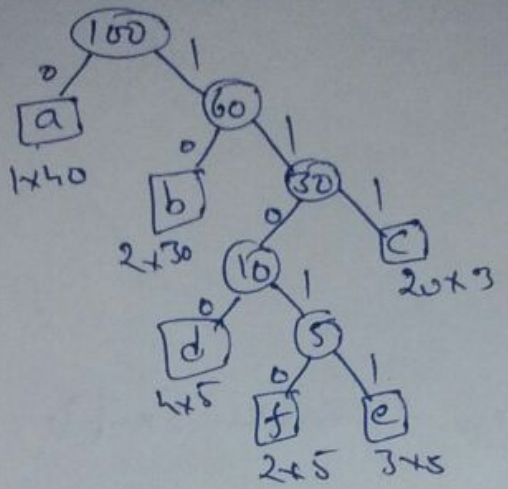
a - 1 - 1
b - 01 - 2
c - 000 - 3
d - 001 - 3

50(1) + 40(2) + 5(3) + 5(3)
50 + 80 + 15 + 15 = 160 bits.

Ex 3

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

Soln



a - 0
b - 10
c - 111
d - 1100
e - 11011
f - 11010

* $\frac{1\times40 + 2\times30 + 20\times3 + 4\times5 \times 2\times5 + 3\times5}{40+30+20+5+3+2}$ } * total no of bit requires

Ex-4

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

✓

Q Suppose the letters a, b, c, d, e, f have probabibilities $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \frac{1}{32}, \frac{1}{32}$

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

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

b) 11, 10, 010, 001, 000

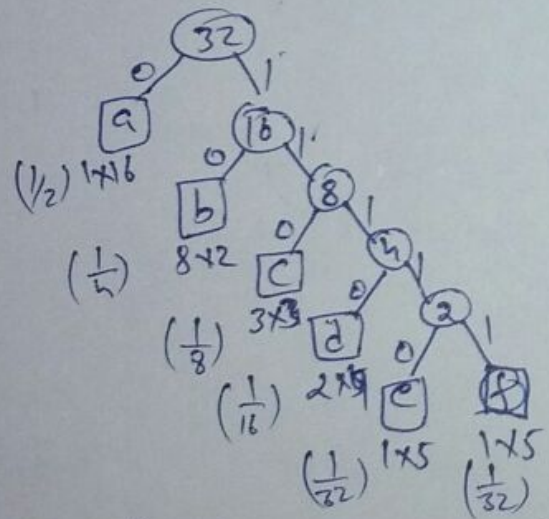
c) 11, 10, 01, 001, 0001, 0000

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

Sln * Try to remove fraction to make simple

So Here multiply by 32 we get

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



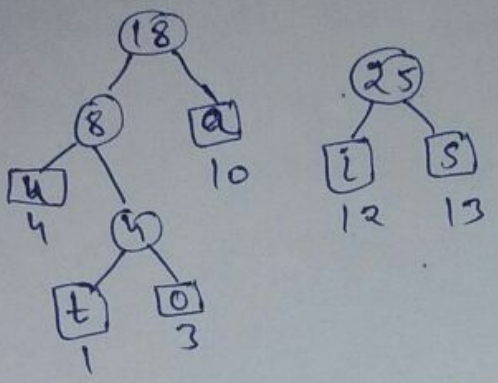
What is the average length of correct ans.

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

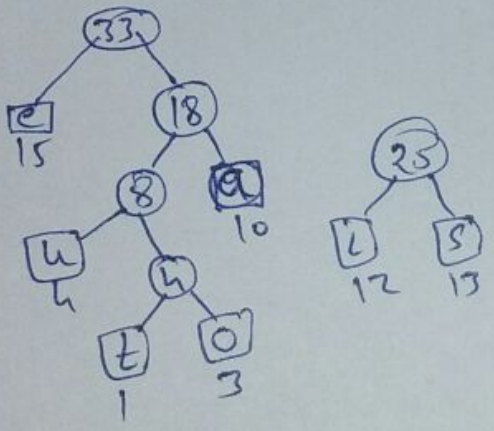
Sln $$\frac{1\times16 + 8\times2 + 3\times3 + 2\times4 + 1\times5 + 1\times5}{16+8+4+3+1+1} = 1.9375$$

or * $$\left(\frac{1}{2}\right)(1) + \left(\frac{1}{4}\right)(2) + \frac{1}{8}(3) + \frac{1}{16}(4) + \frac{1}{32}(5) + \frac{1}{32}(5) = 1.9375$$

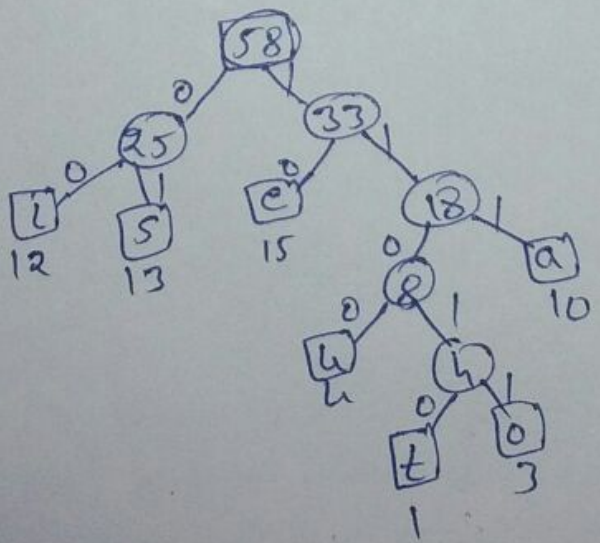
a	e	i	o	u	s	t
10	15	12	3	4	13	1



↓



↓



a - 111
e - 10
i - 00
o - 11011
u - 1100
s - 01
t - 11010

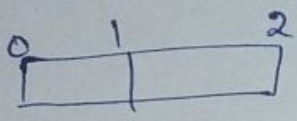
Job sequencing with deadlines

1) create an array of max. job deadline.
2) sort out all the job acc. to max profit
3) start putting job in array
   * Put job in such a way that its deadline be far away.

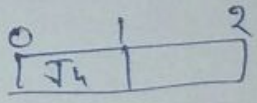
Ex

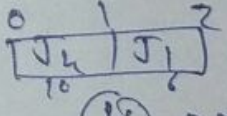
	J_1	J_2	J_3	J_4
D:	2	1	1	2
P:	6	8	5	10

∴ max job deadline = 2



Step-1 find max profit that is 10
then next max profit 8
but 8 can't be put now because its deadline is 1
which is already filled.
find next max profit is 6 put it



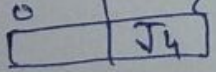


(18) → max profit

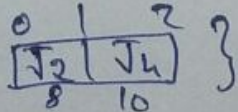
Method
* Put Job at its max deadline

* But its not max we can do it sin more batter way.

Step-1 find max profit that is 10



Now find max is 8
Here we can put J_2 because
its deadline is 1.



→ (18) → max profit

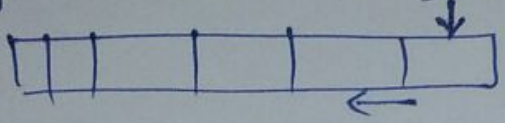
* Time Complexity → $O(n^2)$

HOW?

for sorting job → $O(n \log n)$

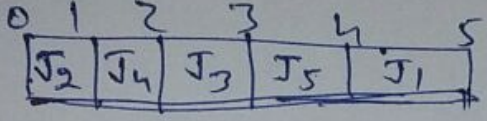
for n job we have to search n slots to put it its right position → $O(n \times n)$

*↳ go to max then start searching



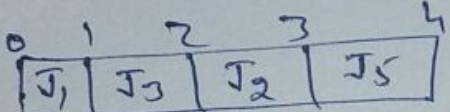
Ex

	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



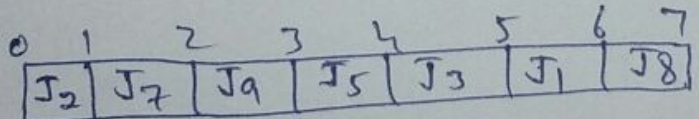
Ex

	J_1	J_2	J_3	J_4	J_5
P:	2	4	3	1	10
D:	3	3	3	4	4



Ex

	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

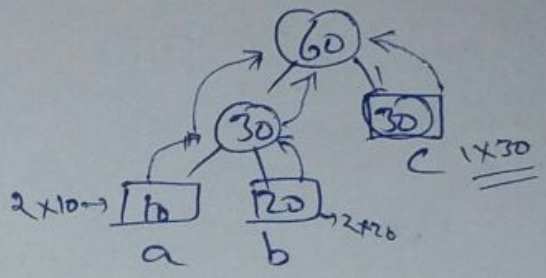


optimal Merge patterns

same like Huffman

Ex!

a	b	c
10	20	30



total no of movement!- 2x10 + 20x3 + 30x1 = 90

Complexity → $O(n^2)$

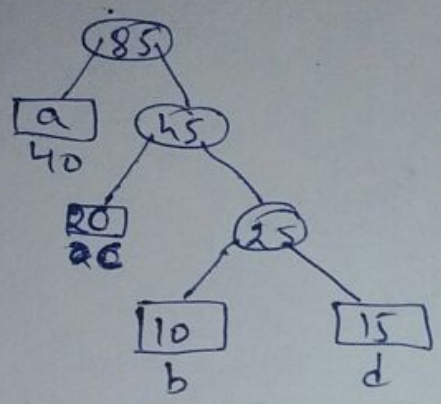
How?

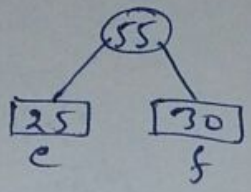
min heap → O(n)

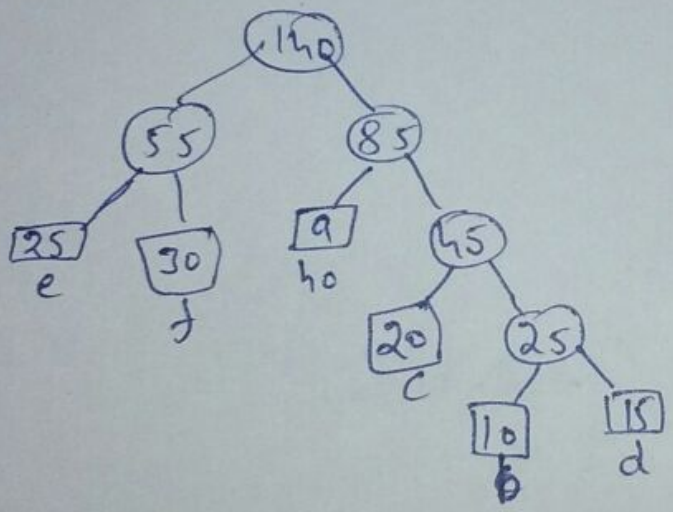
2min → O(logn)
insert → O(logn) } → O(nlogn)

Qx!

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







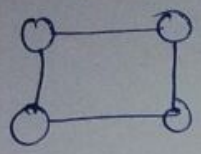
total record movement!-

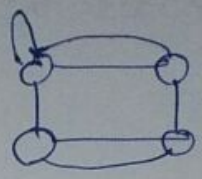
a - 2×40 - 80
b - 4×10 - 40
c - 3×20 - 60
d - 4×15 - 60
e - 2×25 - 50
f - 2×30 - 60

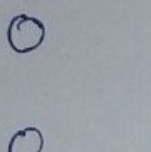
350

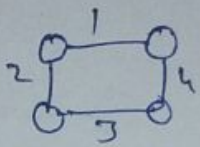
Graphs

(Vertices, Edges)

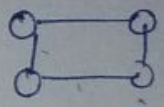
 → simple graph

 → multigraph

 → Null graph



→ weighted graph

 unweighted graph

* Mini no of edges in simple graph = 0

* Max. no of edges = $^nC_2$ = $\frac{n(n-1)}{2}$

*[Here n is vertices]

* No of Edges = $\frac{n(n-1)}{2} = \frac{n^2-n}{2} = O(n^2)$

$E = O(V^2)$.

Apply log on both side

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

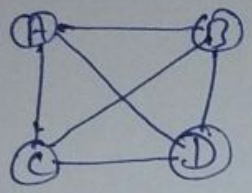
* No of vertices → $V = O(\log E)$

* No of Edges → $E = O(V^2)$

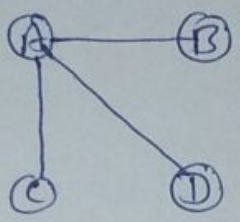
Spanning Tree

Mini no of edges present in a graph which connect all nodes of graph.

Ex:-

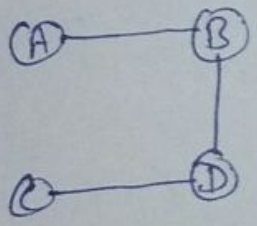


Spanning tree :-



→ 3 Mini edge required to connect.

or



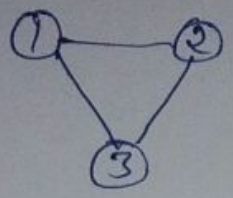
& we have more way to connect this graph.

* for 'n' nodes we ~~require~~ mini no of edges require to connect is 'n-1'.

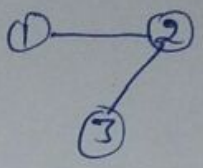
* 'v' edges 'v-1' edges require.

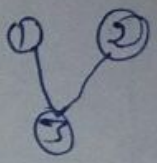
* A spanning tree also called SubGraph.

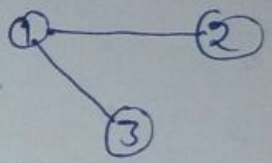
Ex labled Graph :- no having name called labled.



→ spannis tree







★ If there are n node then (n^{n-2}) ~~edg~~ spanning tree possible in complete graph only

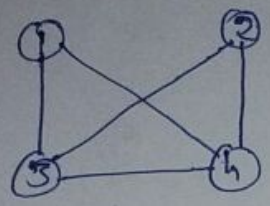
Ex:- n = 3

$ST = (3^{3-2}) = 3$

↙ Spanning tree

No of spanning tree in incomplete graph.

Kirchoff Theorem



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

Kirchoff rule

1) Diagonal 0's replace with degree of node.
   * Degree of node is no of edges incident on that node.
2) Non-diagonal is replace with (-1)
3) Non-diagonal 0's replace with 0's or means remain same.

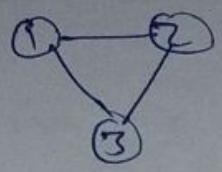
	1	2	3	4
1	2	0	-1	-1
2	0	2	-1	-1
3	-1	-1	3	-1
4	-1	-1	-1	3

* ST = Cofactor of Element

$$\text{Cof}(1,1) = 2(9-1) - (-1)(-3-1) + (-1)(1+3)$$
$$= 8$$

* If a Complete graph degree of node is $(n-1)$

Ex!-

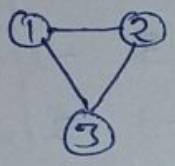


→ 3 node

$n = 3$

Deg. $= 3-1 = 2$

Ex!1



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

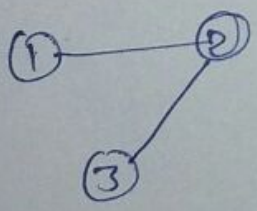
→

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

→ $4-1 = 3$

Spanning tree

Ex!2



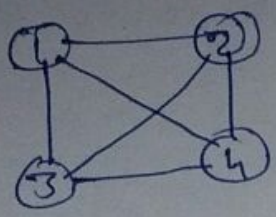
	1	2	3
1	0	1	0
2	1	0	1
3	0	1	0

→

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

→ $2-1 = 1$

Ex 3



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

→

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

$3(9-1)-(-1)(-3-1)+(-1)(1+3)$

$= 24 - 8 = 16$

2

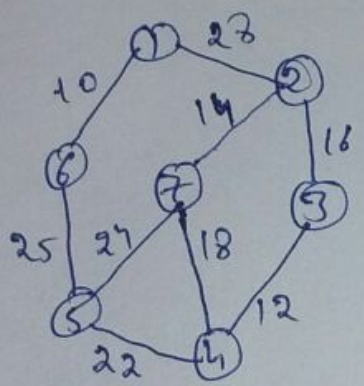
3

→ Its fall in Greedy problem → two way → Prims
↳ kr

Mini. spanning Tree & Prims Algo

Here we can have more than one spanning tree but we will take spanning tree with less or mini cost.

Prims

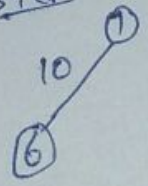


Step1 Take Edge with mini cost

Step2 then next nearest node which make less cost edge.

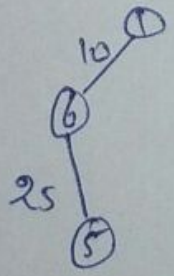
Step3 No cycle should form.

Step-1

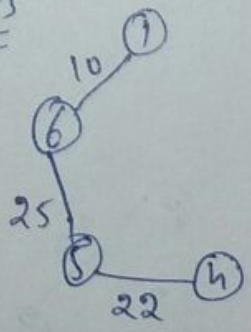


Now see (1,6) which can make less cost edge
* Here 6 – 5 has 25 cost.

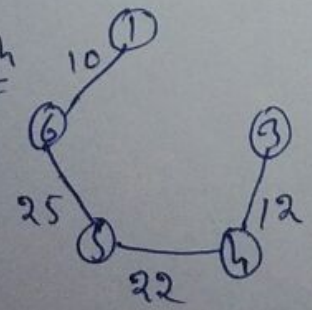
Step-2



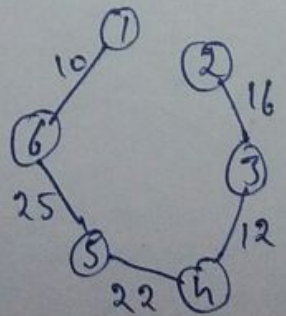
Step-3



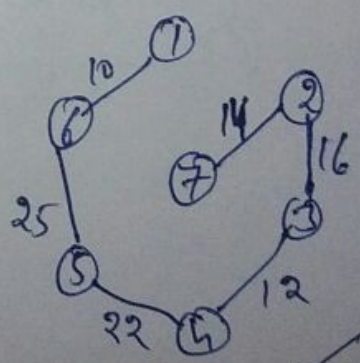
Step-4



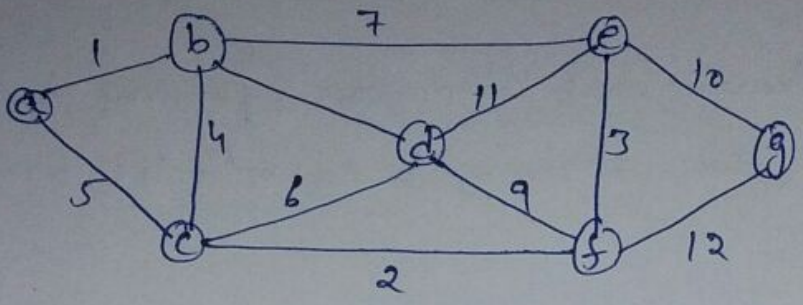
→

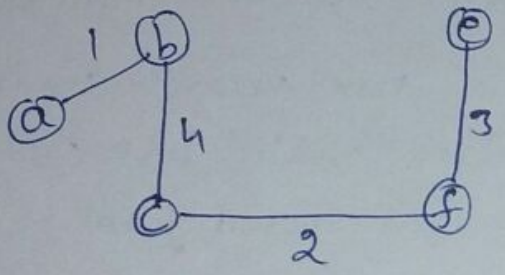


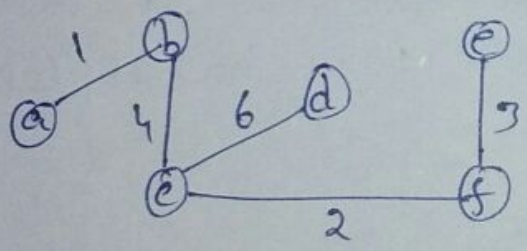
→

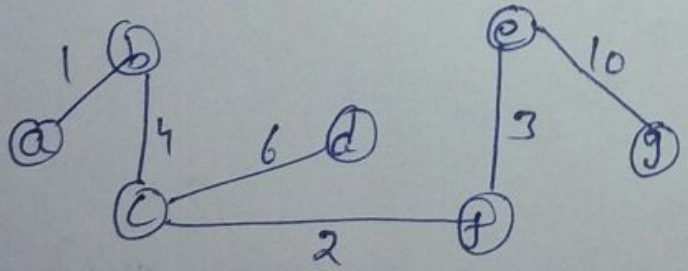


Ex 2


a
b
c
d
e
f
g
1
7
10
11
4
3
5
6
9
12
2

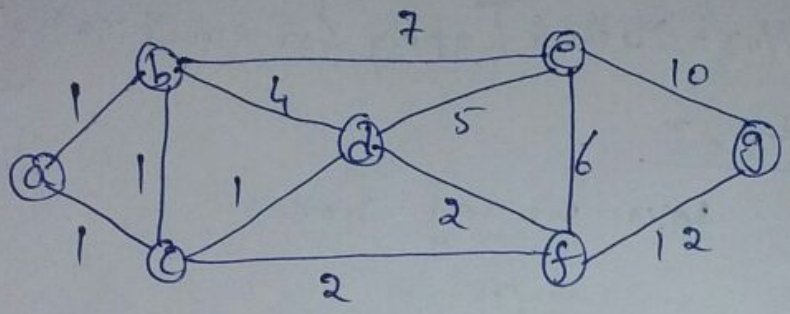

a
b
c
e
f
1
4
3
2


a
b
c
d
e
f
1
4
6
3
2

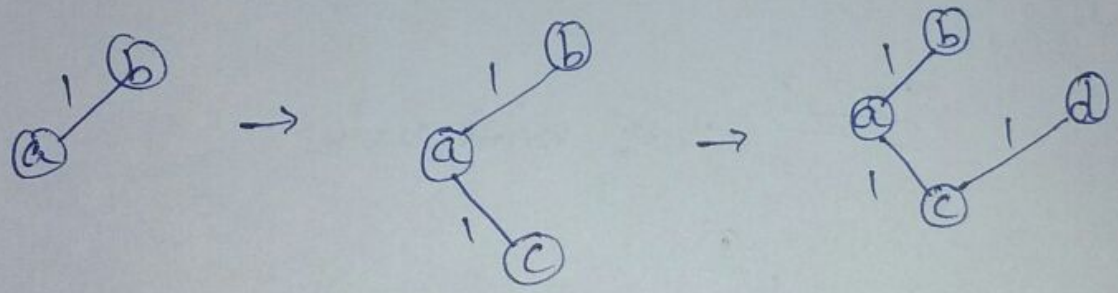

a
b
c
d
e
f
g
1
4
6
3
10
2

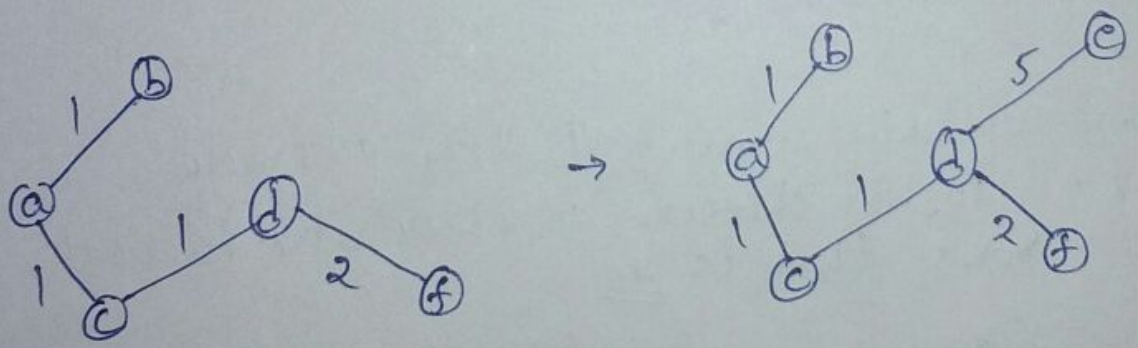
Ex.3

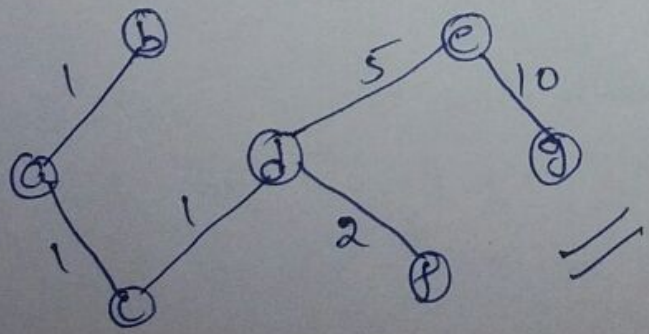
* Whenever we have same weight for more than one edges, might chance of getting more than one spanning tree but cost of those will same.



* Can take any mini edge



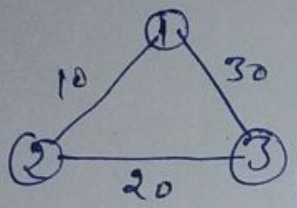


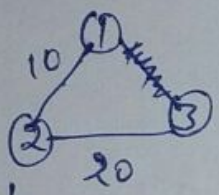


Constructing Graph from Matrix!-

Ex!

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



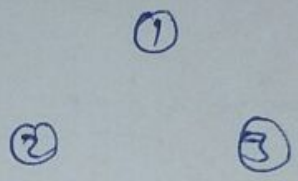


Min ST

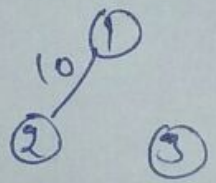
SHort Method [apply for small Matrix]

Step-I

take no of nodes

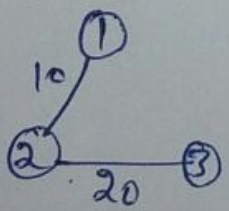


Step-2 check mini value



Step3 Now we have connect (1) & (2) so check row 1 & 2, 10 has choosen so check max. in both rows. that is 20

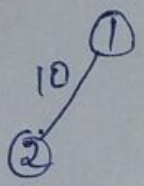
	1	2	3
1	0	(10)	30
2	(10)	0	20
3	30	20	0



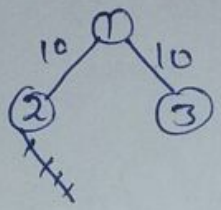
Ex 2

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

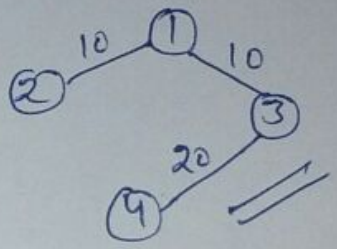
Step1



Step2



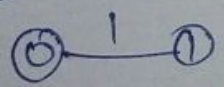
Step3



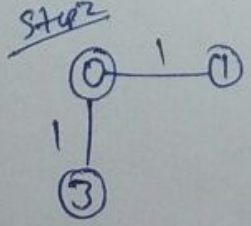
Ex 3

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

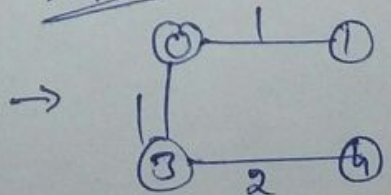
Step1



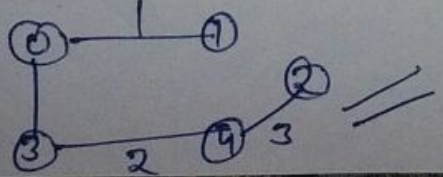
Step2



Step3



Step-4 Now only one Node remaining that is 2 so see column 2 & find mini. [∴ 4-2 is 3]



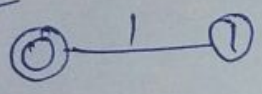
1-10

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

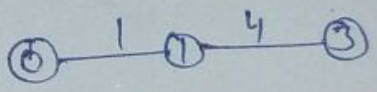
Q. Construct Mini spanning tree such as 0 is a leaf.

[∵ 0 as leaf means 0 should not have more than one edge]

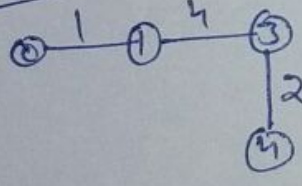
Step-1



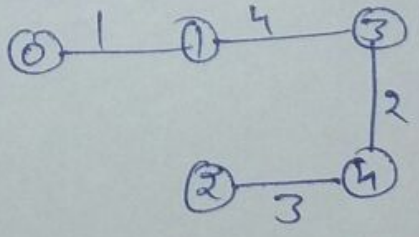
Step-2 Now mini is 0 to 3 but we can't add this so choose next mini



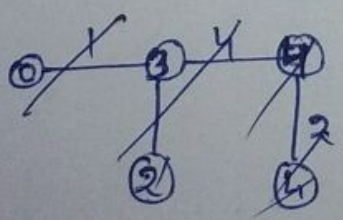
Step-3

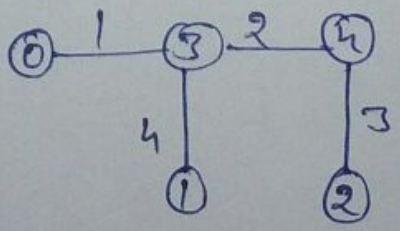


Step-4 Now only one Node remain 2 see column & find mini.



* Second Way



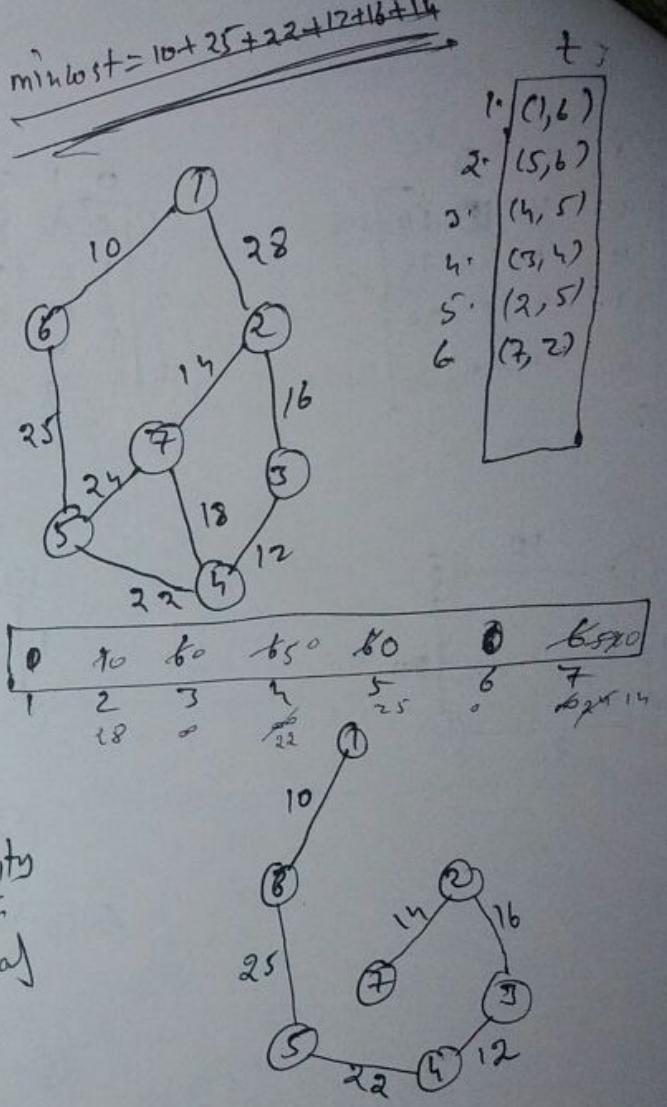


* Prims algo implementation without min heap

```
Algorithm Prim(E, cost, n, t)
// E is the set of edges. cost is (n×n) adjacent 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;  — O(E)
2. mincost = cost[k,l];
3. t[1,1] = k; t[1,2] = l;
4. for (i=1 to n)  — O(n)
5.   if (cost[i,l] < cost[i,k] then near[i] = l;
6.   else near[i] = k;
7. near[k] = near[l] = 0;
8. for (i=2 to n-1)  — (n-2)
   {
9.   let j be an index such that near[j] ≠ 0 and
     cost[j, near[j]] is minimum;  — O(n)
10.  t[i,1] = j; t[i,2] = near[j];
11.  mincost = mincost + cost[j, near[j]];
12.  near[j] = 0;
13.  for k=1 to n do  — O(n)
14.    if ((near[k] ≠ 0 & (cost[k, near[k]] > cost[k, j]))
         then near[k] = j;
   }
}
```

* Prims algo takes $O(n^2)$ time Complexity

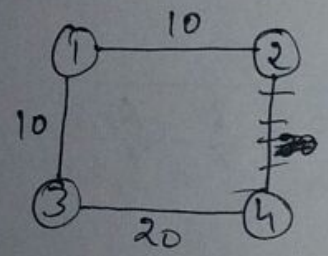
Here is no of vertices

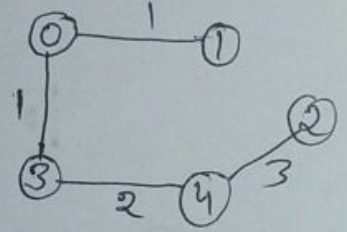


	1	2	3	4
1	0	10	30	50
2	10	0	40	30
3	10	0	0	20
4	50	40	20	0

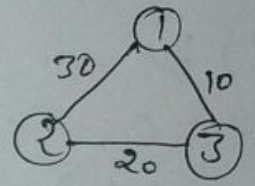
	0	1	2	3	4
0	0	1	8	1	4
1	1	0	12	4	9
2	8	12	0	7	7
3	1	4	7	0	2
4	4	9	3	2	0

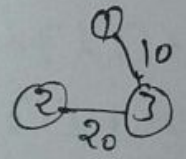
	1	2	3
1	0	10	30
2	10	0	20
3	30	20	0





(7) —

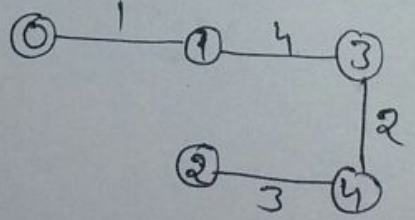




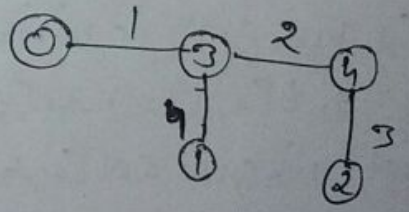
= (30) ✓

(40) ~~=~~

Construct graph as 0 is leaf (means) 0 have 1 edge.



or

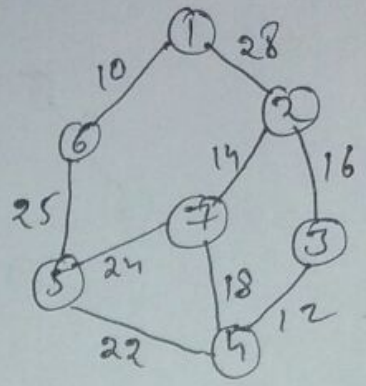


(10) ✓.

(5) =

* Prim's algo Implement using min heap

```
MST-PRIM(G, cost, r) //using min heap
{
for each vertex u ∈ G.vertices } -O(V)
{
  u.key = ∞
  u.π = NIL
}
r.key = 0
Q = G.vertices  -O(V)
while(Q ≠ ∅)  -O(V)
{
u = Extract-min(Q)  -O(logV)
for each vertex 'v' adjacent to 'u'
{
if v ∈ Q & Cost(u,v) < v.key
v.parent = u
v.key = w(u,v)  -O(logV)
}
```



Dense graph - $E = O(V^2)$

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

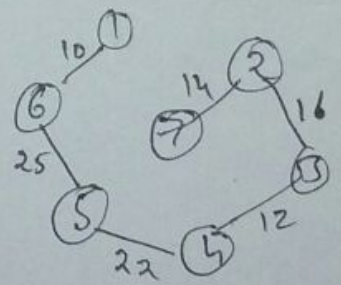
sparse - $E = O(V)$

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

↓ with heap

↓ without heap

	1	2	3	4	5	6	7
key	0	∞	∞	∞	∞	∞	∞
π	N	N	N	N	N	N	N



Extract-min - $O(V \log V)$

Decrease key - ~~$O(V \cdot V \log V)$~~ $= O(E \log V)$

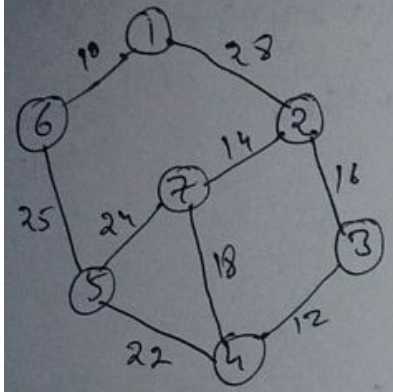
Build heap - $O(E)$

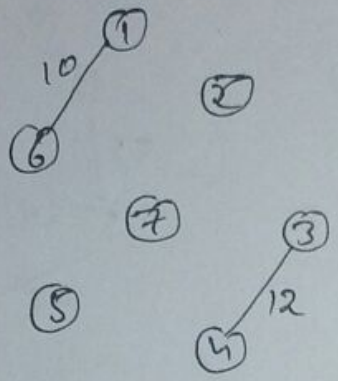
$T.C = O(V \log V + E \log V + E) = \boxed{O(E \log V)}$ // when use min heap

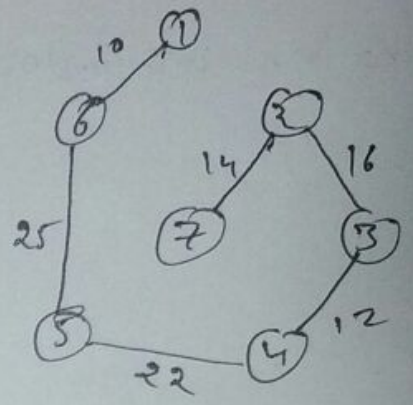
$O(V \log + E)$ // when use fibonacci heap.

$\rightarrow O(V^2)$ // when use no heap (without heap)

Introduction to kruskal algo.







* It select mini. edge. It can like this
* Don't make cycle.

GATE-2000

Let 'G' be an undirected connected graph with distinct edge weight. Let maxe be the edge with maximum weight & mine be the edge with mini. weight which of the following is false?

a) Every mini. spanning tree of 'G' must contain mine.

b) If maxe is in a minimum spanning tree, then its removal must disconnect 'G'.

c) No minimum spanning tree contains maxe.

d) 'G' has a unique mini. spanning tree.

GATE-7

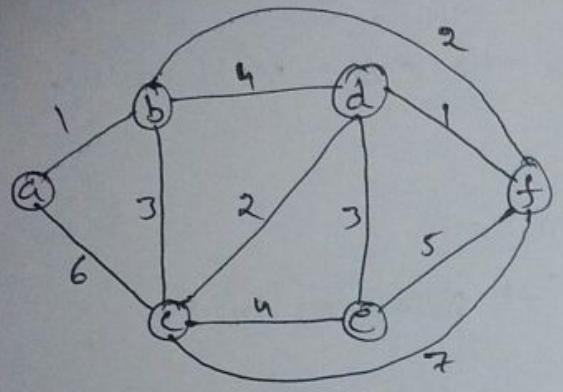
Let 'w' be the mini. weigth among all weights in an undirected connected graph. Let 'e' be a specific edge of weight 'w'. Which of the following is FALSE.

a) There is a mini. spanning tree containing 'e'.

b) 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.

c) Every mini spanning tree has an edge of weight 'w'.

d) 'e' is present in every mini. spanning tree.



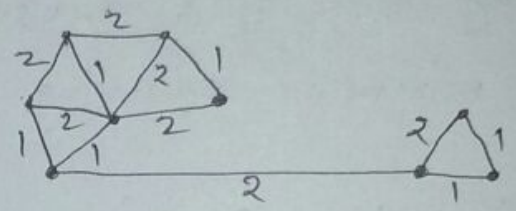
Which of the following can't be the sequence of edges added in that order to mini spanning tree using Kruskals algo.

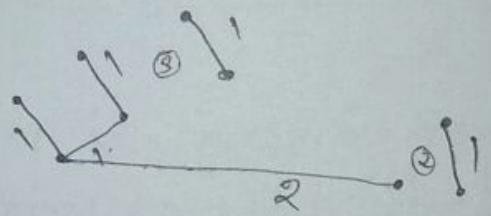
a) (a-b) (d-f) (b-f) (d-c) (d-e)
b) (a-b) (d-f) (d-c) (b-f) (d-e)
c) (d-f) (a-b) (d-c) (b-f) (d-e)
d) (d-f) (a-b) (b-f) (d-e) (d-c)

Ans → d

GATE-14

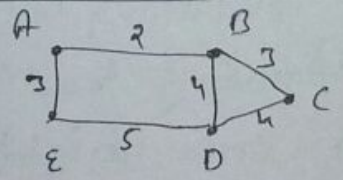
The no of distinct mini. spanning tree for the weighted graph below is

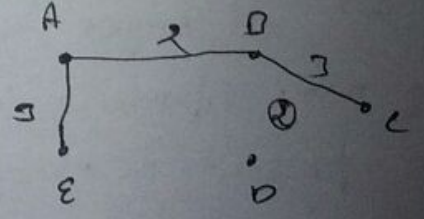




$2 \times 3 = (6)$

Ex





$= (2)$

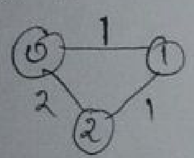
GATE-14

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

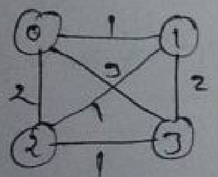
a) The weight of edge (u,v) is |u-v|.

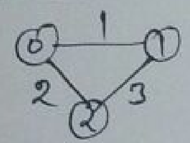
b) The weight of the edge (u,v) is (u+v).

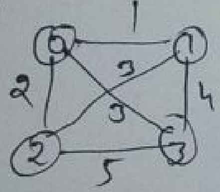
let n=3

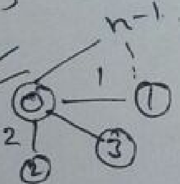


n=4

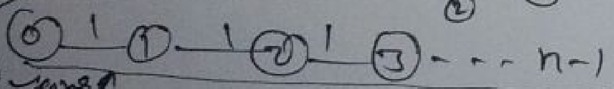








ans (b)



a) ans of a's

* Amortized Analysis → PLEASE REFER PAGE NO 8 TO SEE This topic
AND
Aggregate Analysis →

Ex. of Aggregate Analysis is at Page No 8 Back Side.

* Data structure for disjoint sets

A disjoint-set data structure maintains a collection $S = \{S_1, S_2 \cdots, S_k\}$ of pairwise disjoint dynamic sets.

i.e, if S_i & S_j, $i \neq j$, are two sets, then there is no element that is in both S_i & S_j

$\Rightarrow S_i \cap S_j = \phi$, $i \neq j$

*

Operation on set → Please visit on Page 83-87 in c Programming.

Dijkstra Algo ⚠ Not Apply for -ve weights

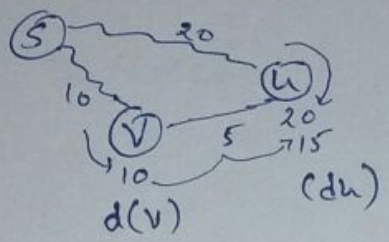
* Not apply for -ve values

* Sometime works sometime not thats why we not apply for -ve weight

Method use in dijkstra

relaxing an Edge :-

Ex



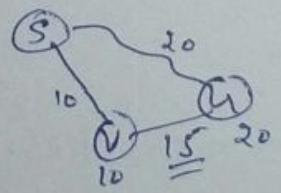
At S to V we get value 10
S to u we got 20

* But we see from S → V → u we get 15
So here we will relax 5 value edge.

[∵ relaxing edge means if we use this edge in path reduce cost.]

* It is not show always we get this shortest path by relaxing it in this case we will not do relax.

Ex



Here will not relax 15 it incr. cost.

Also

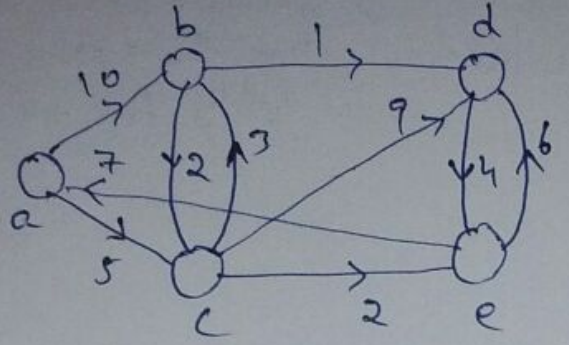
Relaxing an edge (V, u)

if d(u) > d(v) + c(u, v)

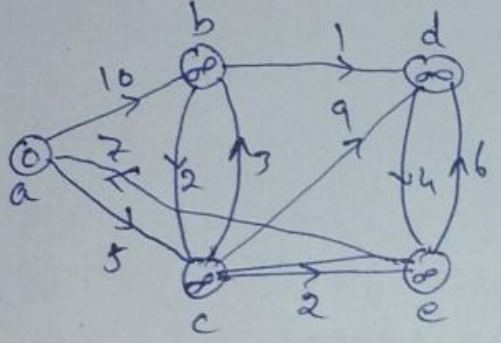
d(u) = d(v) + c(v, u)

Time Complexity (O(1))
in heap (Elogn)

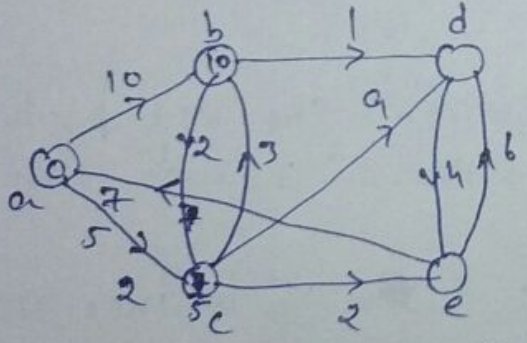
* ~~Shortest Path~~ single source Also [Dijkstra] shortest path



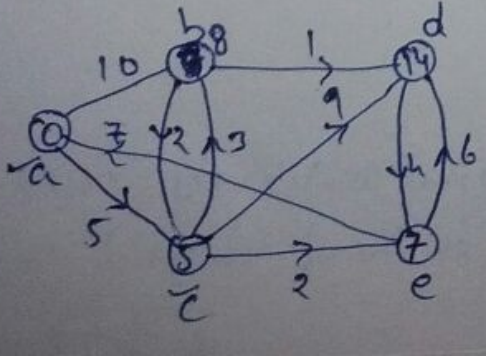
Step-1 start from a -> so a=0, & rest will ∞



Step-2 relax every edge outgoing from a.



Step-3 Now see ~~edge~~ vertex which have least weight Here 5 so relax all outgoing edge.

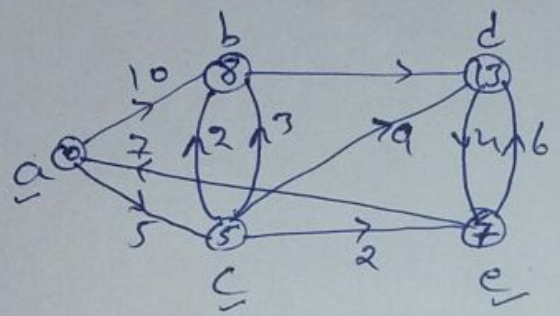


Step-4

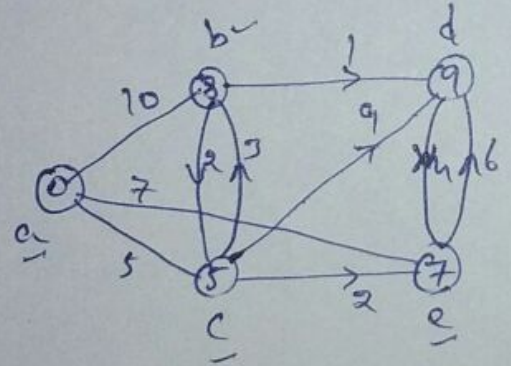
Note:! once a node or vertex is relaxed will not relax again. Till Now a & c are relaxed.

=

Now see the vertex with least value. that is e → 7.
Here least value is c → 5 but its already relaxed so we will not do again.
So relax e



Step-5 Now b has least weight



Step-6 Now only one node remain d is least value.
So No need to relax its already have least value.

So Shortest Path got

Diykstra Algo

Diykstra (G, w, s)
1. Initialize - Single - Sourse (G, s)
2. $S = \phi$
3. $Q = G.V$
4. While $Q \neq \phi$

   u = Extract-min(Q)

   $S = S \cup \{u\}$

   for each vertex $v \in G.Adj[u]$

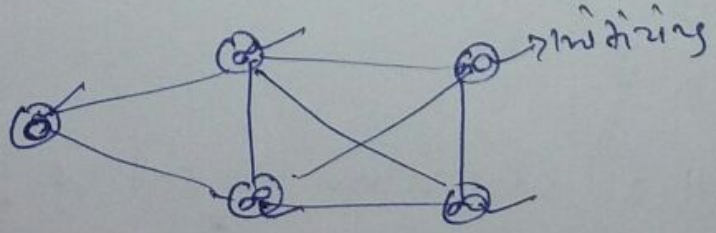
   relax (u, v, w)

Time complexity :- $O(E \log v)$ [∵ When using minheap]

G → Graph - vertex - Edges
w → weight
s → ~~Start~~ Short path

Step1 Initialize sorce in graph

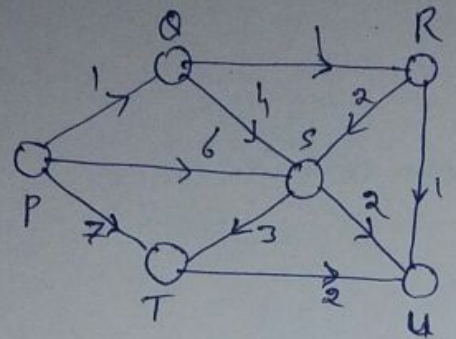
Ex



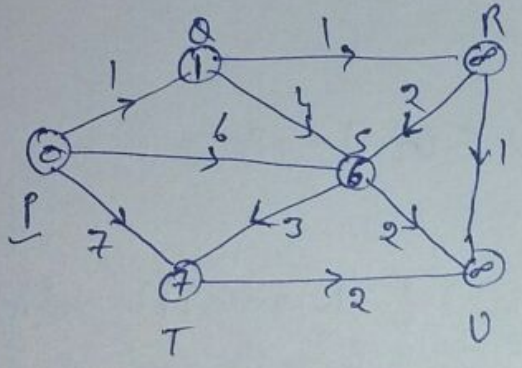
Step2 $S = \phi$
shortest path at first its zero or ϕ

Step3 Q → ~~with~~ build heap.

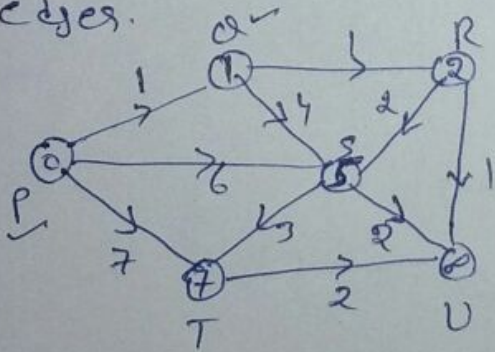
Ex-2



Step1 start from P & relax outgoing edge.



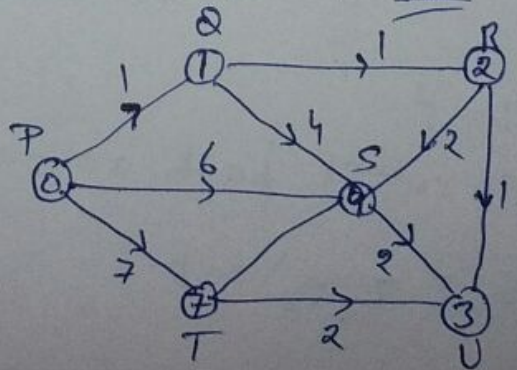
Step2 take least weight vertex that is Q & relax its outgoing edges.



Step-3 Repat steps till all edge relax we got at last.

Path SHortes

PQRUST



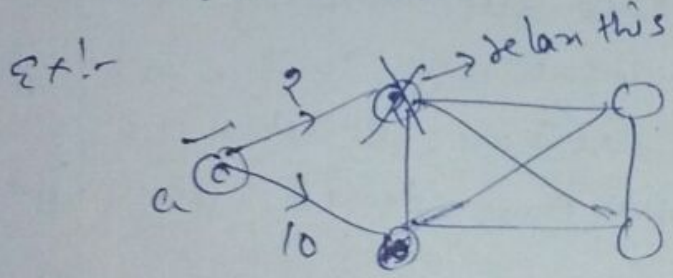
Step-4 Q is not Empty

Step-5 u = Extract min(Q)
take smallest vertex

Step-6 S = S ∪ {u}
add smallest vertex to set of vertex
whose ~~wt~~ shortest weight is already found out.

Step-7 for each vertex v ∈ G Adj[u]
Now relax ~~every~~ ~~edge~~ ~~of~~ whose vertex
weight is less.

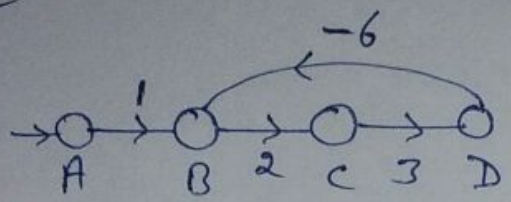
Ex:-



Step-8 Now find again shortest weight (less) vertex
& relax that.
[repeat step 5] till all vertex relaxed.

* Dijkstra fails in -ve edge cycle

Ex:-



Here after Node D its will be a chance we get batter path so also will try to go B but B is already selected.

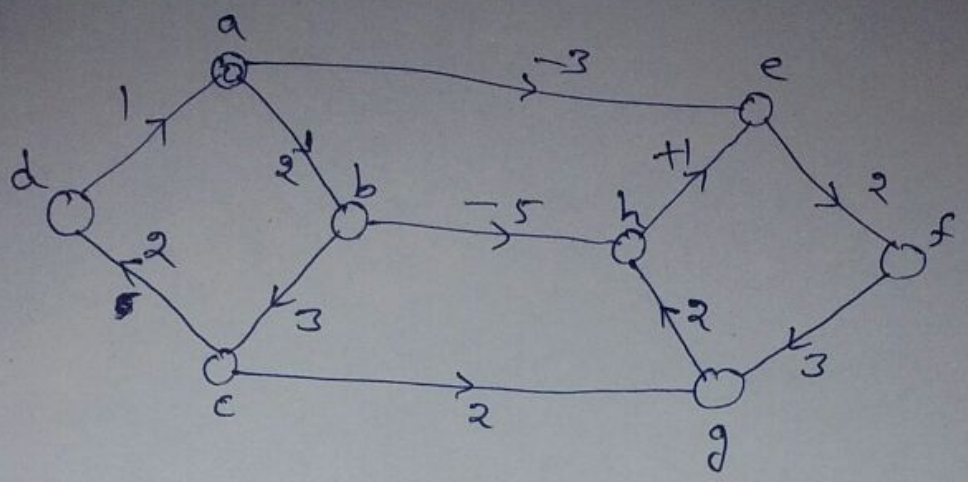
If we think that at D we got 6 & again we go to B value will be 0 at B. Its Batter than 1.

~~But its wrong & if also keep moving on cycle it will go to -ve∞. its wrong so Dijkstra fails.~~

But its wrong once we have selected or relaxed an edge we cannot do it again because its out of heap.

So Here Dijkstra fails because at B we need to reduce value from 1 to 0 because its batter but we can't do so, thats why its fail.

Ex

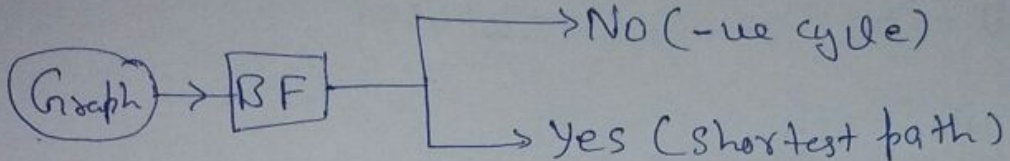


	a	b	c	d	e	f	g	h
	0	∞	∞	∞	∞	∞	∞	∞
a		2	∞	∞	-3	∞	∞	∞
e		2	∞	∞		∞	∞	∞
f		2	∞	∞			2	∞
g		2	∞	∞				4
b			5	∞				-3
h			5	∞				
c				7				
d								

Path we got:- a e f g b h c d

Bellman Ford Algo

Bellman ford algo has capability to say that there is an -ve edge cycle or not.

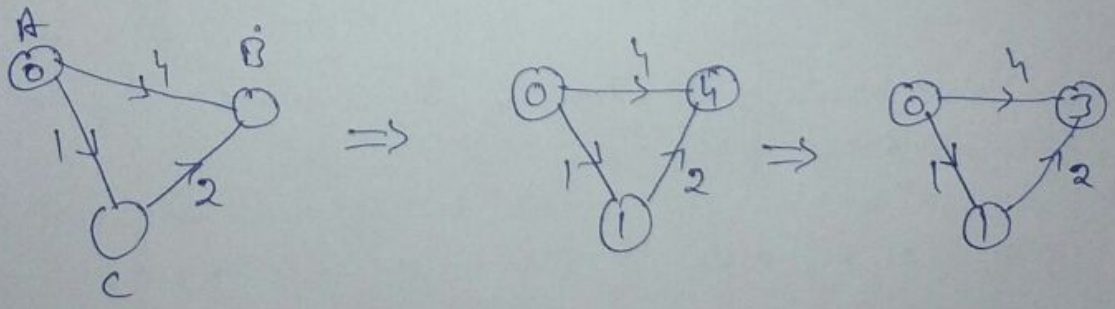


* But time complexity of Bellman ford algo is more as compare to Dijkstra algo.
So for +ve edge or graph which not having -ve edge cycle we apply Dijkstra algo.

How check Bellman that there is a -ve edge cycle?

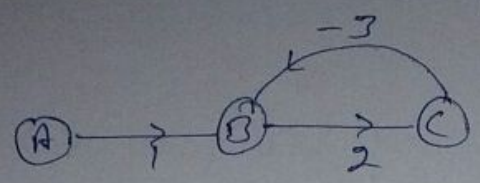
In Bellman after relaxing all edges. Relax edges one more time, if value is decr. that means there is an -ve edge cycle.

Ex:-



* If there are n edges or n vertexs then after relaxing "n-1" time relax one more time "n". If value decr. means -ve cycle exist.
* In Normal graph we relax edges(n-1)times.
Ex:- for 3 edge graph there will one shortest path possible that contains 2 edges.

Ex:-



Time complexity = (VE)

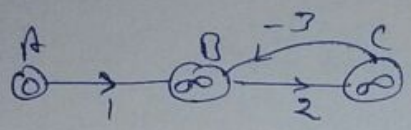
(Vertex. Edges) [Array] used

In Dijkstra = (log) (Elog v)

[∵ Heap used]

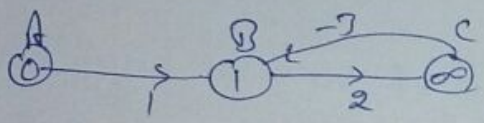
min

Step 1 Relax all edges



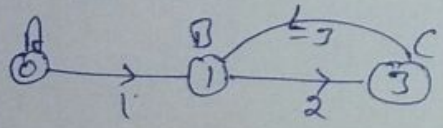
Here we relax A got 0 then relax B got (1+∞) = ∞ so keep same vthen for C (0+2) = ∞ so keep same.

Step 2 Relax again all edges



Here we relax B we got 1 then relax C we got (0+2) = ∞

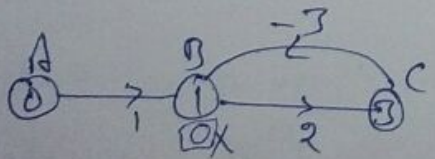
Step 3 Relax again all edges



Now we have 3 vertex so its contain only two edges for shortest path.

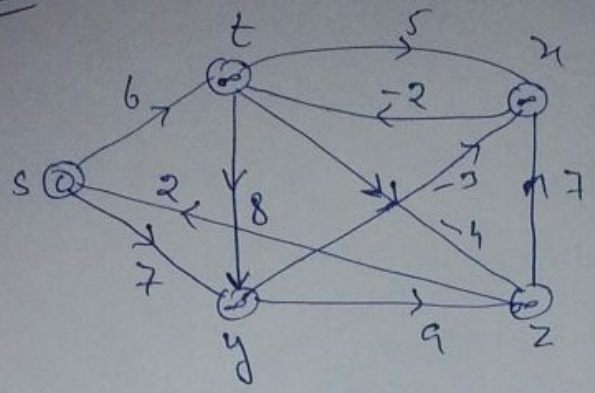
* But Bellman will relax edges once more from here if value ~~so~~ decr. It say -ve cycle exist

Step 4

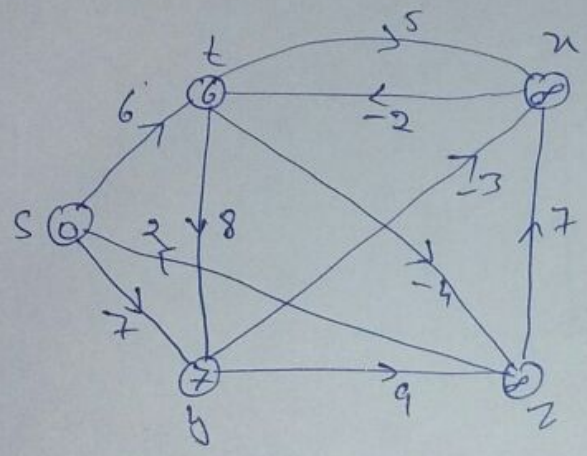


Now after relaxing we got 0 [(+3-3)]. But here we have 1. So ~~here~~ value change after "n-1" round. That means -ve cycle exist.

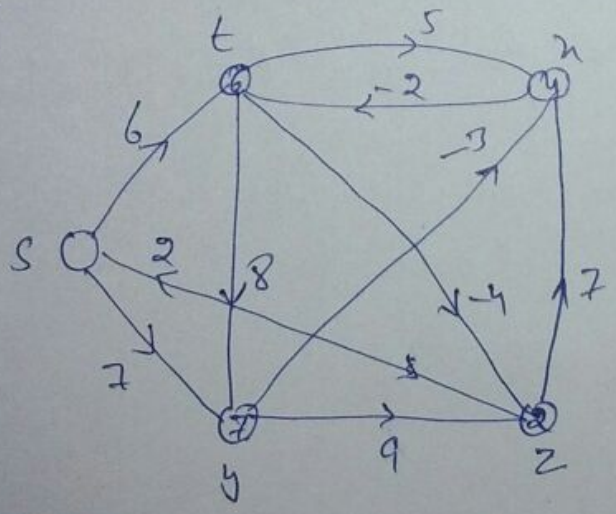
Ex!-



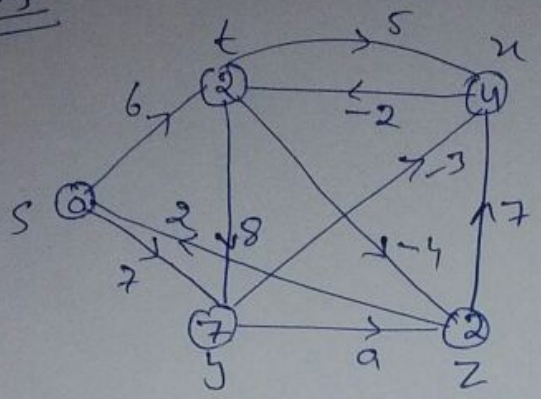
Step 1



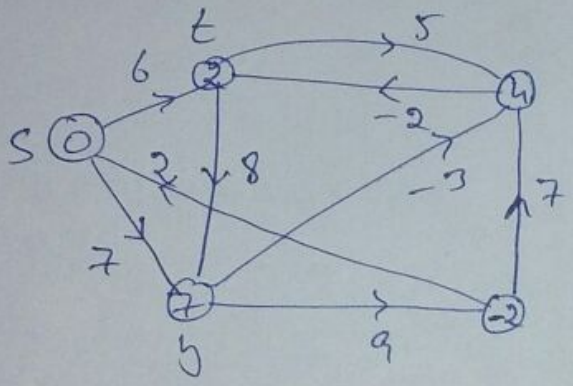
Step 2



Step 3



Step 4



Step 5 There we have 5 vertex so we have to relax 4 time & we got shortest path.

But Bellman will relax one more time to check -ve weight cycle.

Here we will got same ans so there is no -ve weight cycle.

Bellman ford Also

* Time complexity :- O(VE)

```
Bellman-Ford (G, w, s)
{
Initialize single source (G,s)
for i=1 to |G.V| - 1
    for each edge (u,v) ∈ G.E
    RELAX (u,v,w)
for each edge (u,v) ∈ G.E
    if (v.d > u.d + w(u,v))
        return FALSE
return TRUE
}
```

G → Graph (vertices, edges set)
w → $ weight
s → start

Step-1 Initialize single source (G,s)
means start a node & rest node distance will ∞.

Step-2 then first loop will run (n-1) time.

Step-3 second ~~n~~ loop run for each edge

Step-4 Relax edge [∵ Array is used]

Step-5 Take every edge & try to relax one more time
if we get less value of node means there is -ve cycle.
else return True.

DAG (Directed Acyclic Graph)

Directed means - Direction

Acyclic → No cycle

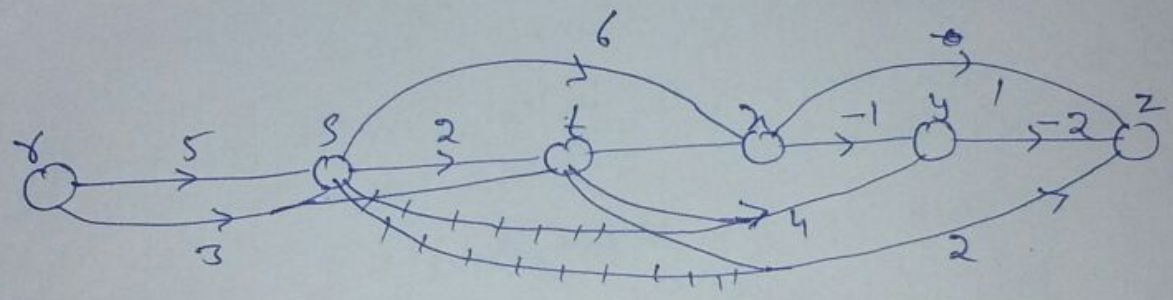
* Topological short used

to apply short time complexity O(V+E)

Topological sort!- go from only left to right

Can't go back.

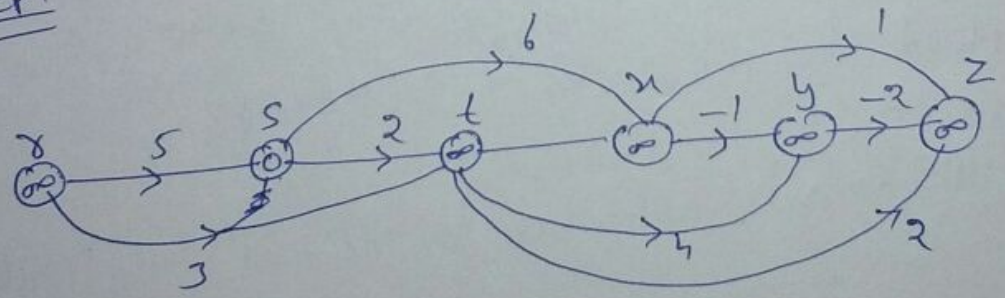
Ex!



* start node is s

Now we find path

Step1

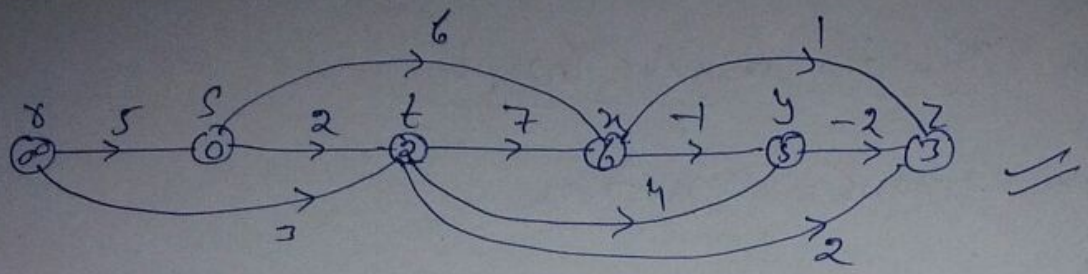


Step2

Relax every edge.

So when we relax r it will remain ∞

— through —



Also

```
Dag-Shortest-paths(G, w, s)
{
1) Topologically sort the vertices of 'G'.
2) Initialize-Single-Source(G, s)
3) for each vertex u, taken in to topologically
   sorted order
4)    for each vertex v ∈ G.adj[u]
5)       Relax(u, v, w)
}
```

* General fastest algo is Dijkstr $O(VE\log v)$
  - Bellman ford $O(VE)$
  - DAG $O(V+E)$ → only aplicable for no -ve weight cycle.

Matrix chain Multiplication (Dynamic Programming)

* A: $p \times q$, B: $q \times r$

* Total no of scalar multiplication require $\boxed{p \times q \times r}$

Ex

A: 2×1, B: 1×2, C: 2×4

Now we have two way to solve this
Either we multiply first & second then there resultant matrix will multiply with C.
or we multiply B & C first then ther resultant with C.

Way 1

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

$2 \times 1 \times 2 = 4$

$2 \times 2 \times 4 = 16$

So total multiplication occur $= 16 + 4 = 20$
& result matrix will size of $[2 \times 4]$

Way 2

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

$1\times2\times4 = 8$

A

$[\ ]_{2\times1}\ [\ ]_{1\times4}$

$2\times1\times4 = 8$

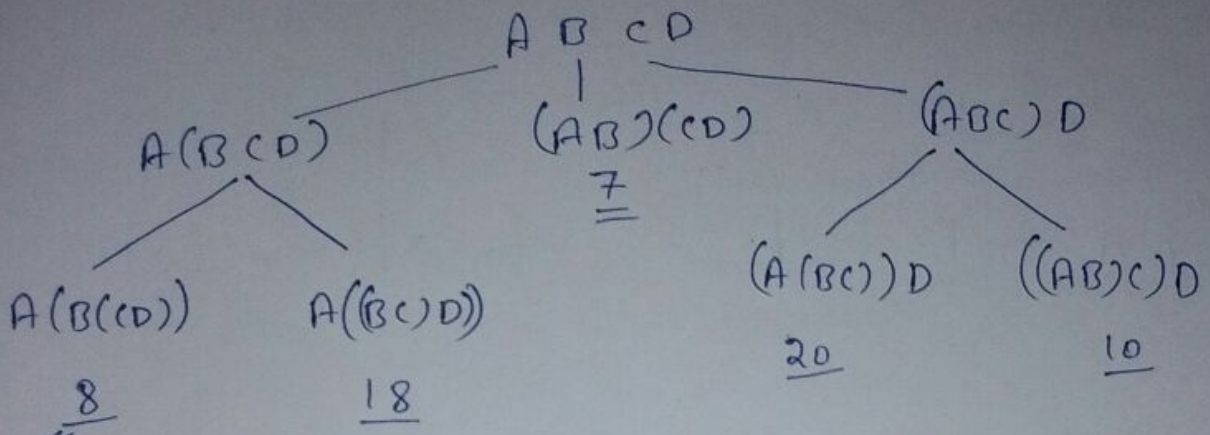
Total multiplication require $= 8+8 = 16$

This is batter than way 1 because here we have to do only 16 multiplication.

Resultant matrix size of $[2\times4]$.

* Either we do by way 1 or way 2 resultant matrix will same.

Ex! for 4 Matrix



total multiplication requires

* Formula to calculate How many way to calculate Matrix

$$\frac{(2n)!}{(n+1)!\,n!}$$

Ex!- let size of Matrix is 5 (ABCDE)
then value of n = 4 [∵ one less than size]

$$\frac{(2(4))!}{(4+1)!\,4!} = \frac{8!}{5!\,4!} = \frac{8 \times 7 \times 6 \times 5!}{5!\,4 \times 3 \times 2} = 14 \text{ ways}$$

Ex!
For 4 [ABCD]
n = 3

$$\frac{(2(3))!}{(3+1)!\,3!} = \frac{6!}{4!\,3!} = \frac{6 \times 5 \times 4!}{4!\,3 \times 2} = 5 \text{ ways}$$

optimal substructure & recursive soln

let we have size of matrix

$A_1 A_2 A_3 \text{ ---- } A_n$

Then their dimention will

$p_0 \times p_1 \quad p_1 \times p_2 \quad p_2 \times p_3 \text{ --- } p_{(n-1)} \times p_n$

Similarly

$A_i \; A_{i+1} \; A_{i+2} \text{ --- } A_j$

→ Now to get optimal we break big problem in small parts.

So these matrix can be ~~to~~ break from A_i, A_{i+1}, A_{i+2} or it can be (A_k).

then.

$$\boxed{(A_i A_{i+1} \text{ --- } A_k)(A_{k+1} \text{ --- } A_j) + p_{i-1} p_k p_j} \rightarrow \text{recursive eqn.}$$

Dimention → $p_{i-1} \times p_k \qquad p_k \times p_j$

for $i \le k < j$

Recursive Eqn:

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

Zero when there is only one matrix so no need of multiplication.

Dynamic Programming (in Multichain matrix multi
(Bottom UP Implementation)

Total no of function call $\boxed{\frac{n(n+1)}{2}}$

Ex:- A_1 A_2 A_3 A_4
$p_0\,p_1$ $p_1\,p_2$ $p_2\,p_3$ $p_3\,p_4$

(1,1) (2,2) (3,3) (4,4) — 4
(1,2) (2,3) (3,4) — 3
(1,3) (2,4) — 2
(1,4) — 1
10 ✓

∴ ~~n=~~

10(10+1)

∴ $\begin{cases} n = 4 \end{cases}$

$\frac{4(4+1)}{2} = 10$ ✓

Algo of Bottom up Dy. Programing

```
MATRIX-CHAIN(p)
{
1.  n = p.length - 1
2.  Let m[1...n, 1...n] and 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]
            m[i,j] = q
12.
13.         s[i,j] = k
14. return m and s
```

Ex!

A_1	A_2	A_3	A_4
1×2	2×1	1×4	4×7
$p_0\ p_1$	$p_1\ p_2$	$p_2\ p_3$	$p_3\ p_4$

Time complexity = $O(n^3)$

Space cond = $O(n^2)$

(1 1) 0	(2 2) 0	(3 3) 0	(4 4) 0
2 (1 2)	8 (2 3)	4 (3 4)	
(1 3) 6	(2,4) 6		
(1,4) 7			

(+2)

$(1\,2) = 1\times2\times1$

$(2\,3) = 2\times4$

$(3\,4) = 1\times4\times1$

$$(1\,3) = \begin{cases} (1\,1) + (2,3) + p_0 p_1 p_3 = 16 \\ (1,2) + (3,3) + p_0 p_2 p_3 = 66 \end{cases}$$

$2 \quad 0 \quad + \quad 8\ 4$

$$(2,4) = \begin{cases} (2\,2) + (3,4) + p_1 p_2 p_4 = 6 \\ (2,3) + (4,4) + p_1 p_3 p_4 = 16 \end{cases}$$

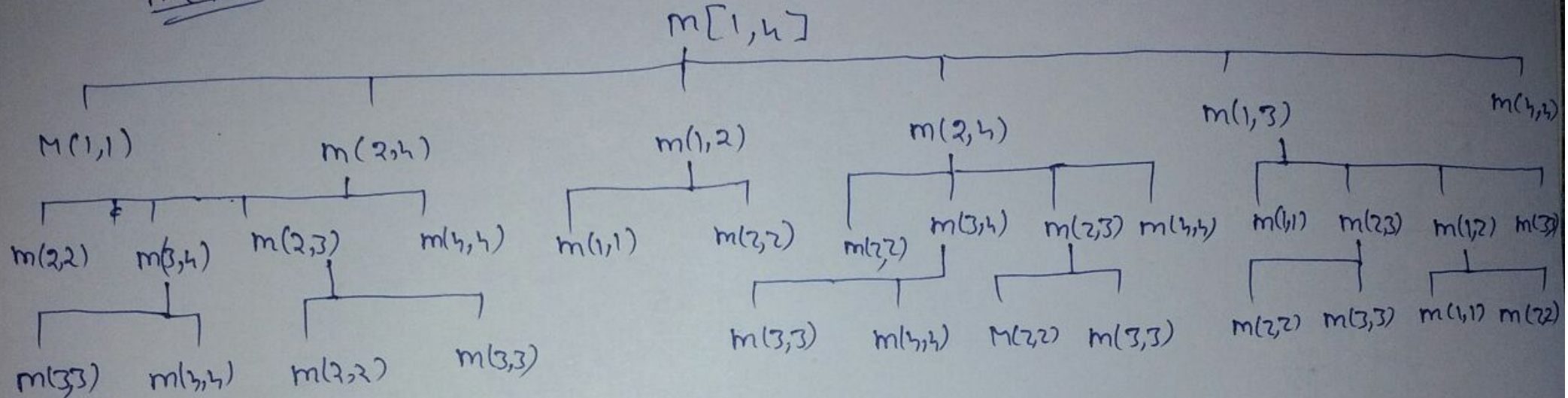
$8 \quad 0 \quad 8$

$$(1,4) = \begin{cases} (1,1) + (2,4) + p_0 p_1 p_4 = 8 \\ (1,2) + (3,4) + p_0 p_2 p_4 = 7 \\ (1,3) + (4,4) + p_0 p_3 p_4 = 8 \end{cases}$$

Recursion Tree

Ex:-

m[1,4]



* Here Some values are repeting. thats why we use Dynamic programming to remove repetation (recursion).

** Time Complexity by recursion Tree = (2^n) → exponential.

* Time Complexity by Dynamic Programming $O(n^3)$

* Top down dynamic programming memoization algo

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

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

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

```
lookup-chain(m, p, i, j)
{
1. if m[i,j] < ∞
2. return m[i,j]
3. if i == j
4. m[i,j] = 0
5. else for k = i to j-1
6. q = lookup-chain(m, p, i, k) + lookup-chain(m, p,
        k+1, j) + p_{i-1} p_k p_j
7. if q < m[i,j]
8. m[i,j] = q
9. return m[i,j]
```

GATE-11

$(3-1) = 2$ split

500
×50
40000

M_1 (10×100), M_2 (100×20), M_3 (20×5), M_4 (5×80)

11 0	22 0	33 0	44 0
12 20000	23 10000	34 8000	
13 15000 ← '1'	24 50000 → '3'		
14 19000 → '3'			

split happen

$$(1,3) = \begin{cases} (1,1) + (2,3) + 5000 = 15000 \checkmark \\ \quad 0 \qquad\quad 10000 \\ (1,2) + (3,3) + 10000 = 21000 \\ \quad 20000 \qquad 0 \end{cases}$$

$$(2,4) = \begin{cases} (2,2) + (3,4) + 160000 = 168000 \\ \quad 0 \qquad\quad 8000 \\ (2,3) + (4,4) + 40000 = 50000 \checkmark \\ \quad 10,000 \qquad 0 \end{cases}$$

$$(1,4) = \begin{cases} (1,1) + (2,4) + 80,000 = 130000 \\ \quad 0 \qquad\quad 50000 \\ (1,2) + (3,4) + 16,000 = 44000 \\ \quad 20000 \qquad 8000 \\ (1,3) + (4,4) + 40000 = 19000 \checkmark \\ \quad 15000 \qquad 0 \end{cases}$$

$((1(2\,3))4)$ ✓

19000 Ans.

Longest Common Subsequence

String :- set of all possible value

Subsequence :- set of all possible value in incr. order

Ex :- ANKIT

String :→ {AN} {ANK} {KIT} {NKI} {IT}

Subsequence :- {ANKI} {AKT} {NI} {ϕ} {AT}

* for 'm' lngth string '2^m' substring possible

DNA (A, C, G, T)

Ex :-

D_1 = AGCCTCAGT

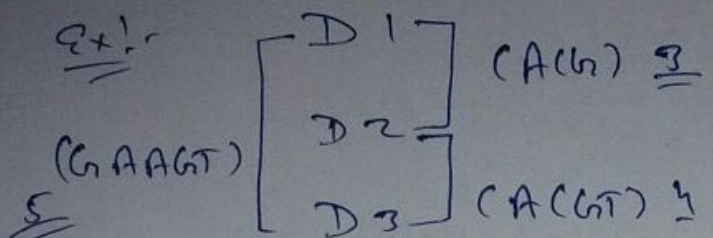
D_2 = GCCT

1) if both DNA equal means there parent is same.

2) ~~if substring then~~
if one DNA is substring of other then they are much similar.

3) find the all possible ~~common~~ subseq. & find longest Comman subseq among them to check How much similar DNA are.

Ex:-



So we can see D1 & D3 are much similar.

Step1 Find all subsequence of 'A'

Step2 Find for each subsequence whether it is a subseq 'B'.

Step3 Find the longest among common subsequence.

Ex:- D1 = ANKIT D2 = ANKYADAV

S_1 = {ANKT} — longest common subsequence

S_2 = {A}

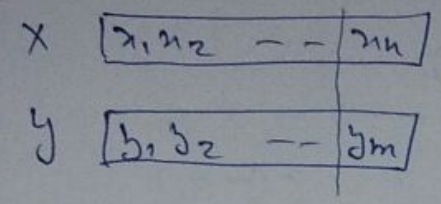
S_3 = { }

Complexity → $(n\,2^m)$

2^m → no of subsequence possible

n → compare subsequence

#optimal substructure



Here we match x & y. let say xn & ym matched then we found 1 subsequence Now we find possible subsequence in remaining string

$x_1, x_2 -- x_i$

$y_1, y_2 -- y_i$

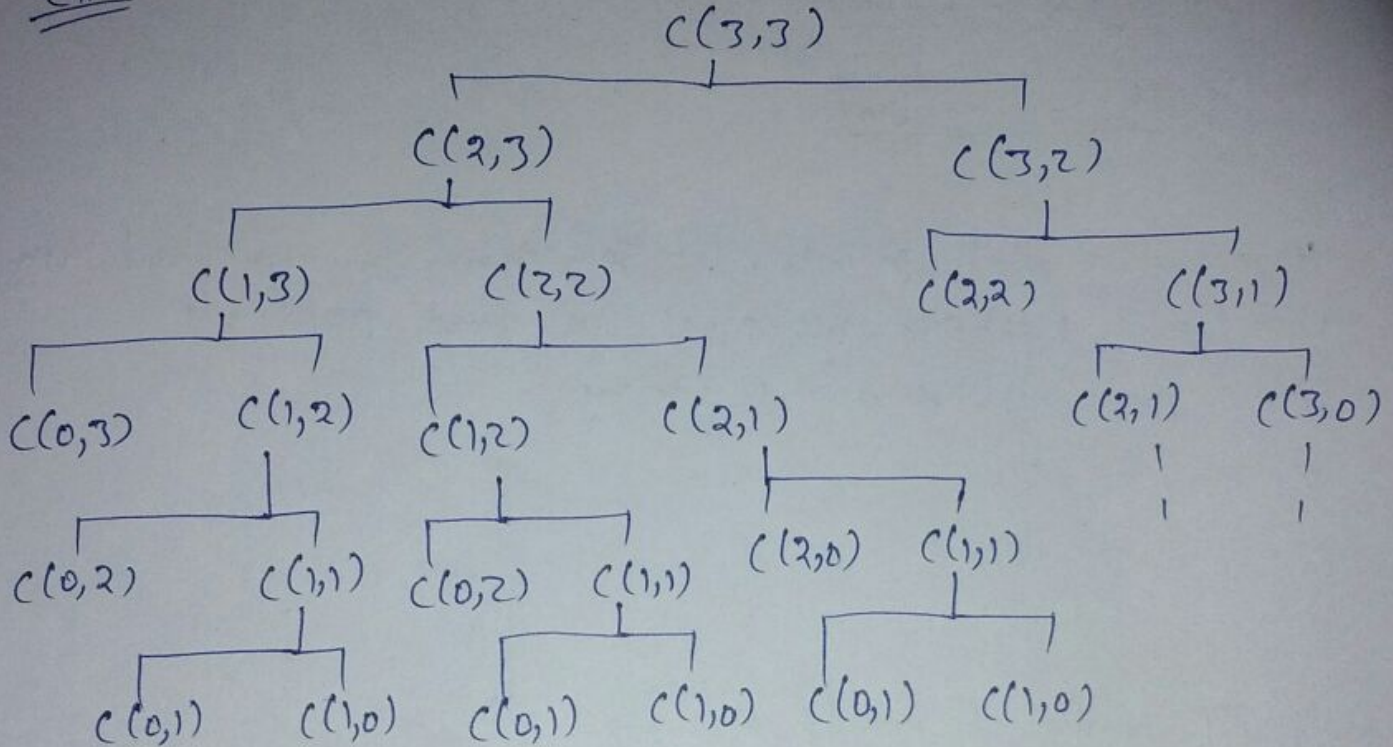
$$c[i,j] = \begin{cases} 0, & i=0, j=0 \\ 1 + c[i-1, j-1]; & i,j>0 \;\&\; x_i = y_j \\ \max(c[i-1, j], c[i, j-1]); & i,j>0 \;\&\; x_i \neq y_j \end{cases}$$

* $c[i-1, j]$ → if $x_i \neq y_j$ then remove last character from i & find subsequence in j.

$c[j-1, i]$ → if $x_i \neq y_j$ then remove last character from j & find subseq in i.

Recursion Tree & unique subproblem

Ex!-



O(n+m) Tree depth →

([n, m]
([n-1, m]
([n-1, m-1]
([n-2, m-1]
([n-2, m-2]
([0,1]
([1,0]

$O(2^{n+m})$

Complexity.

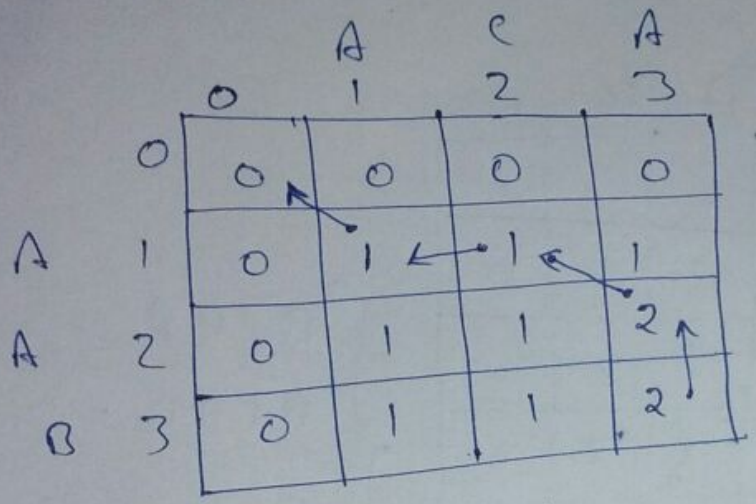
In Dynamic programming

Complexity = O(m×n)

Ex!-

$x = (A\ A\ B)$

$y = (A\ (A)$



(AA)

if match → 1 + c(i-1, j-1)

Not match → max (c[i-1, j], c[i, j-1])

Ex:- $x = \{A, B, C, B, D, A, B\}$

$y = \{B, D, C, A, B, A\}$

	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 —

BDAB —

BCAB —

Bottom up dynamic Programming Also in LCS

```
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 xi == yj
            c[i,j] = c[i-1,j-1] + 1
        else
            c[i,j] = max(c[i-1,j], c[i,j-1])
return c
}
```

Ex:- 14

$A = qpqrr$

$B = pqprqrp$

A/B		q	p	q	r	r
B	0	0	0	0	0	0
p	0	0	1	1	1	1
q	0	1	1	2	2	2
p	0	1	2	2	2	2
r	0	1	2	2	3	3
q	0	1	2	3	3	3
r	0	1	2	3	4	4
p	0	1	2	3	4	4

{ pqrr, qpqr, qprr }

Q

we are given two sequences x[m] & y[n] of length m & n, respectively with indices of x & y starting from 0

L(i,j) is LCS of x[m] & y[n] is given below

L[i,j] = 0, if either i=0 or j=0

= expr1, if i,j>0 and x[i-1] = y[j-1]

= expr2, if x[i-1] ≠ y[j-1] and i,j>0

Q L(i,j) is computed by DP using an array L[m,N] where m=m+1 & N=n+1, such that L[i,j] = L[i,j]

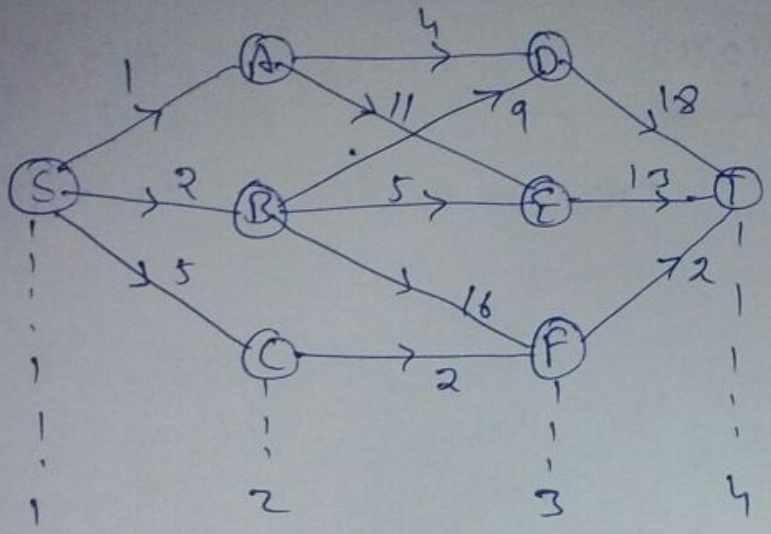
a) all elements of L should be initialized to '0'.

b) The values of L[i,j] may be computed in Rmo & cmo.

c) The values of L[i,j] can't be computed in either Rmo & cmo.

d) L(p,q) needs to be computed before L(r,s] if either p<r or q<s

#Multistage Graph



* No direct connection ~~from~~ from A to B, B to C & D to E, E to f.

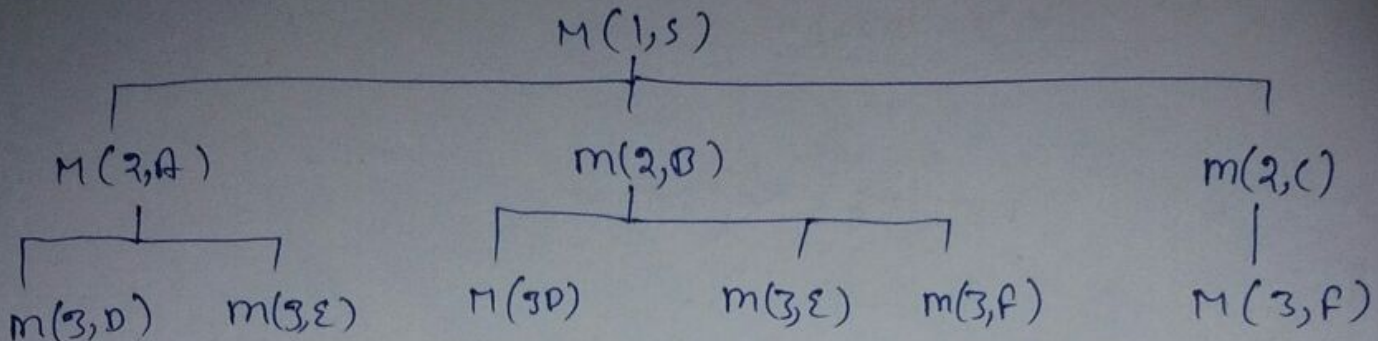
#Recursive Eqn

Now we can break Graph into small subproblems such that from S to we can go A, B & C.

So we find (or break) path from (A to T) (B to T) and (C to T).

Eqn

$$m(1,S) = \min \begin{cases} S \to A + m(\overset{\text{stage}}{\overset{\nearrow}{2}}, A) \\ S \to B + m(2, B) \\ S \to C + m(2, C) \end{cases}$$



Hex complexity will

At depth of tree = k

~~Levels~~

Node at Each Level = N

$= O(k^n)$ → Not Good

So apply dynamic programming.

(Dynamic Programming on MSG)

Ex!- for same Graph

Steps We apply bottom up Dynamic Programming.

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

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

$T[8] = \{ cost(8,8) + T[8] \quad (0 + 0)$

$T[7] = \{ cost(7,8) + T[8] = 2 \quad (2 + 0)$

$T[6] = \begin{cases} cost(6,7) + T(7) & (\infty) \\ cost(6,8) + T(8) & (13 + 0) \end{cases}$

$T[5] = \begin{cases} cost(5,6) + T[6] & (\infty) \\ cost(5,7) + T[7] & \\ cost(5,8) + T[8] & (18) \end{cases}$

$T[4] = cost(4,7) + T[7]$

$T[3] = \begin{cases} cost(3,5) + T[5] = 27 & (9 \;\; 18) \\ cost(3,6) + T[6] = 18 & (5 \;\; 13) \\ cost(3,7) + T[7] = 18 & (16 \;\; 2) \end{cases}$

$T[2] = \begin{cases} c(2,5) + T[5] = 22 & (4 \;\; 18) \\ c(2,6) + T[6] = 24 & (11 \;\; 13) \end{cases}$

$T[1] = \begin{cases} c(1,2) + T[2] = 23 & (1 \;\; 22) \\ c(1,3) + T[3] = 20 & (2 \;\; 18) \\ c(1,4) + T[4] = 9 & (5 \;\; 4) \end{cases}$

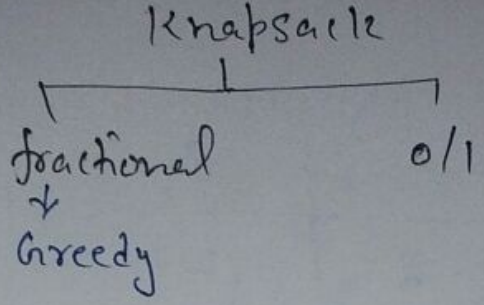
Time Complexity = $1+2 \cdots (n-1) = O(n^2)$

$= O(n^2)$

$= O(V^2)$

$= O(E)$ ✓

0/1 knapsack

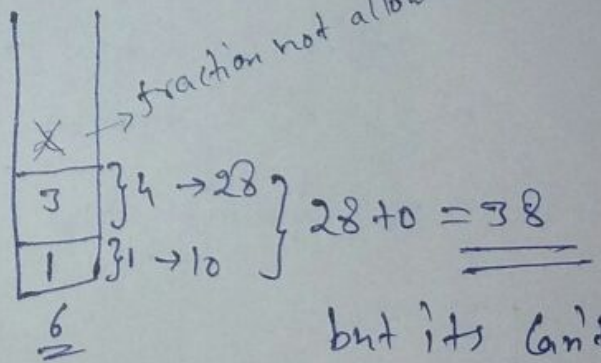


0/1 knapsack :- fraction is not allowed. (Either element will be taken completly else left).

Ex:

capasity = 6

obj.	1	2	3
Prof.	10	12	28
Wei.	1	2	4
P/W	10	6	7



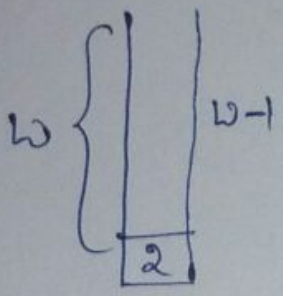
but its can't be best soln.

Note :- So in 0/1 knapsack greedy fails.

* Why Greedy fails for 0/1 Knapsack

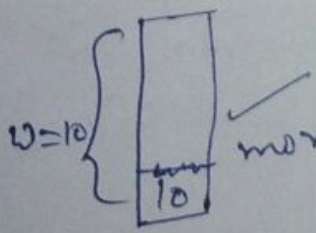
Ex:-

Ob	1	2
Prof.	2	W
weig.	1	W
P/W	2	1



∴ W is variable here not constant

* Now let W is equal to 10
In this condition also Greedy will take obj.1 first acc. to P/W. But we can get max profit by putting W or 10. So Greedy fails here.



Recursive Eqn:-

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

∴ W is weight or capacity of bag.

∴ i is the no of objects

In recursive eq.

Step1: Either we include obj. in bag then

$p_i + KS(i-1, W-w_i)$
- p_i → profit
- $i-1$ → remaining obj.
- $W-w_i$ → remaining capacity

Else we not include obj. then

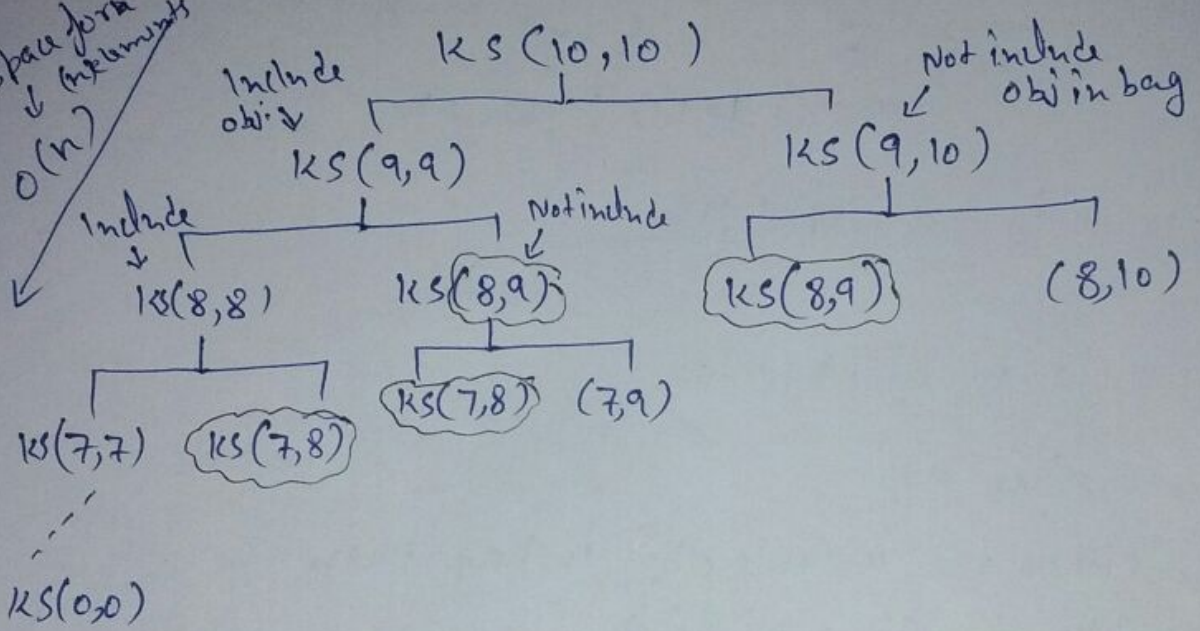
$KS(i-1, W)$
- $i-1$ → remaining obj.
- W → weight or capacity will same in this case because we don't include object.

Step2: we Get 0 profit if no of object remaining d bag is still having space. or we Get 0 profit if bag get full but we have obj. remaining.

Step3: No profit will get when (remaining) obj. weight is more than capacity of bag.

Recursion Tree !-

Ex!- Let KS(10, 10) (n, W) [∵ let weight of each obj is 1-] then

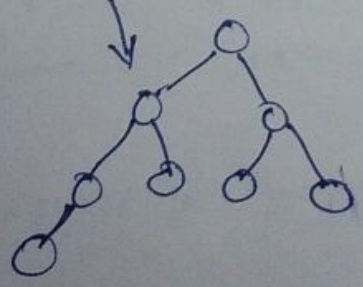


Note!- Here some terms are repeating there might be chance that by applying dynamic programming we can reduce time complexity. [$O(2^n) \rightarrow O(nW)$]

* Time complexity of this tree will $O(2^n)$.

* if tree is not full binary then complexity will $O(2^{n/2}) \cong O(2^n)$.

Ex!-



* No of unique term will in tree $(n \times W)$ [∵ 10×10]

Bottom-up Dynamic Approach!-

Ex!- $c = 6$

0	1	2	3
w	1	2	4
p	10	12	28

* Time & space complexity for this will $O(nW)$.

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

Soln

$KS(n=1, W=1)$

$\{ p_i^{10} + KS(0, 0^{0}) \rightarrow 10$

$KS(0,1)^{0}$

$KS(n=1, W=2)$

$\{ p_i^{10} + KS(0,1)^{0}$

$KS(0,2)^{0}$

* $KS(3,2)$ → bag capacity 2 but obj. 3 weight is 4 so no inclusion

$\{ p_i + KS(2X \rightarrow$ out of capacity so

$KS(2,2)_{12} \rightarrow 12$

$KS(2,2)$

$\{ p_i^{12} = + KS(1,0)^{0}$

$KS(1,2)_{10} = 12$

$KS(2,3)$

$\{ p_i^{12} + KS(1,1)^{10}$

$KS(1,3)_{10}$

$= 22$

$KS(3,6)$

$\{ p_i^{28} + KS(2,2)^{12} \rightarrow 40$

$KS(2,6)_{22}$

NOTE:-

Here we get time complexity is $O(nW)$ is batter than (2^n) but for large value of W we can get agin exponetial time complexity.

Ex:- if value of W will (2^n) then complexity will. . . .

$O(n(2^n))$ → exponention

∴ In this condition it is batter to apply bruteforce method. [∵ try all possible values]

Till Now Dynamic method is batter no, algo is introduced batter than it.

Algo of Dynamic Programming!-

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) and (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

$\{a_1, a_2, a_3 \ldots a_n\}$

find the weight 'w' is possible or not.

* So for 'n' no of elements
  No of subset possible is 2^n
  because each in element have two choice either to include or either not include.

Recursive Eqn! -

$$SS(i,s) = \begin{cases} SS(i-1,s); & s < a_i \\ SS(i-1, s-a_i) \vee SS(i-1,s); & s > a_i \\ \text{True}; & s = 0 \\ \text{false}; & i = 0, s \neq 0 \end{cases}$$

SS → Subset
i → element in array
S → Sum.

false! - when there is no element in array but sum is not zero then condition become false (means we can't get Sum.

True! - when Sum is equal to 0.

$\Rightarrow$ SS(i-1, s-ai)

$\{a_1, a_2 \ldots a_{i-1}\}$ $\{\underbrace{\quad}_{s-a_i}\}$

Ex!- $\{1,2,3,4\}$ $s=10$

Here after ~~removing~~ or selecting last 'i' element
$10-4 = \underline{6}$

remaing element $\{1,2,3\}$ is possible to genrate sum '$s-a_i$' (6)

$\Rightarrow$ SS(i-1), s) ; $s > a_i$

if we don't include 'a_i' then with '$i-1$' element sum 's' is possible or not.

Ex!- $\{1,2,3,4,5\}$ $s=10$

Here not include 5 because its more than sum

$\Rightarrow$ SS(i-1, s) ; $s < a_i$

it possible when 'a_i' is more then sum then no chance that we can include it.

* Time Complexity without DP $\rightarrow 2^n$

Using Dynamic Programing $\rightarrow (nW)$

* if W is large like 2^n then its not good.

Ex!- $\{6, 3, 2, 1\}$

$w = 5$

sum → /	0	1	2	3	4	5
0	T	F	F	F	F	F
1	T	F	F	F	F	F
2	T	F	F	T	F	F
3	T	F	T	T	F	T
4	T	T	T	T	T	T

(0,0) → using 0 element sum 0 possible → yes (T)

(0,1) → using 0 ——— 1 possible → false

(1,0) → using 1st element sum 0 possible → yes because that element is more than required sum

Ex!- $SS(i-1, s)$; $s < ai$

$(1-1, 0) \Rightarrow 0,0 \rightarrow$ True

(1,1) → using 1st element sum 1 possible → false its 6 so not possible.

(2,1) → using 1st two element {6,3} sum 8 1 possible → false

(2,3) → ——— sum 3 possible → True.

Ex!- $SS(i-1, s-ai) \vee SS(i-1, s)$;

$SS(2,3) = S(1,0) \vee (1,3)$

$T \vee F \rightarrow T$

Here using 2nd element remain sum is 0, if dont include element then sum remaining is 3.

(2,4) → $SS(2,4) = S(1,1) \vee (1,4)$

$F \vee F \rightarrow F$

(4,5) → $SS(4,5) = S(3,4) \vee (3,5)$

$F \vee T \rightarrow T$

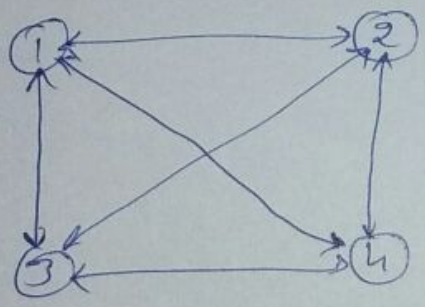
Travelling salesman Problem

[* If we apply brute force Hex then complexity will $(n-1)! = O(n!) \cong (n^n) \Rightarrow >> 2^n$.
This complexity is even bigger than (2^n). & we even don't take this complexity.

optimal Substructure

Weight matrix of Graph

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



$$T(1,\{2,3,4\}) = \min \begin{cases} (1,2) + T(2,\{3,4\}) \to 5 \\ (1,3) + T(3,\{2,4\}) \to 9 \\ (1,4) + T(4,\{2,3\}) \to 7 \end{cases}$$

* Here we start from 1 & we have to reach at 1 by visiting each node once. no cycle should occur.
Now from 1 we have 3 choice that we can go to at 2 or 3 or 4.
So three case will become (1,2) (1,3) (1,4)

When we reach from (1,2) then again we have two choices from 2 to 3 or 4.
from (1,3) we have 2 choices 3 to 2 or 4.
from (1,4) ——— 4 to 2 or 3.

Now we have to calculate $T(2,\{3,4\})$ & so on sub problem

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

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

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

Now Calculate $(3\{4\})$ & so on. sub problem

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

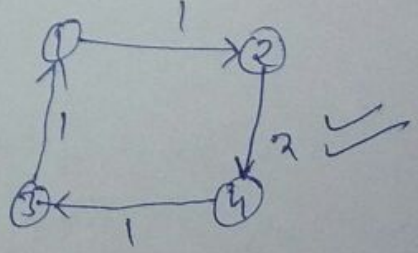
$T(4,\{3\}) = (4,3) + T(3,\phi) = 2$

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

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

$T(2,\{3\}) = (2,3) + T(3,\phi) = 5$

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

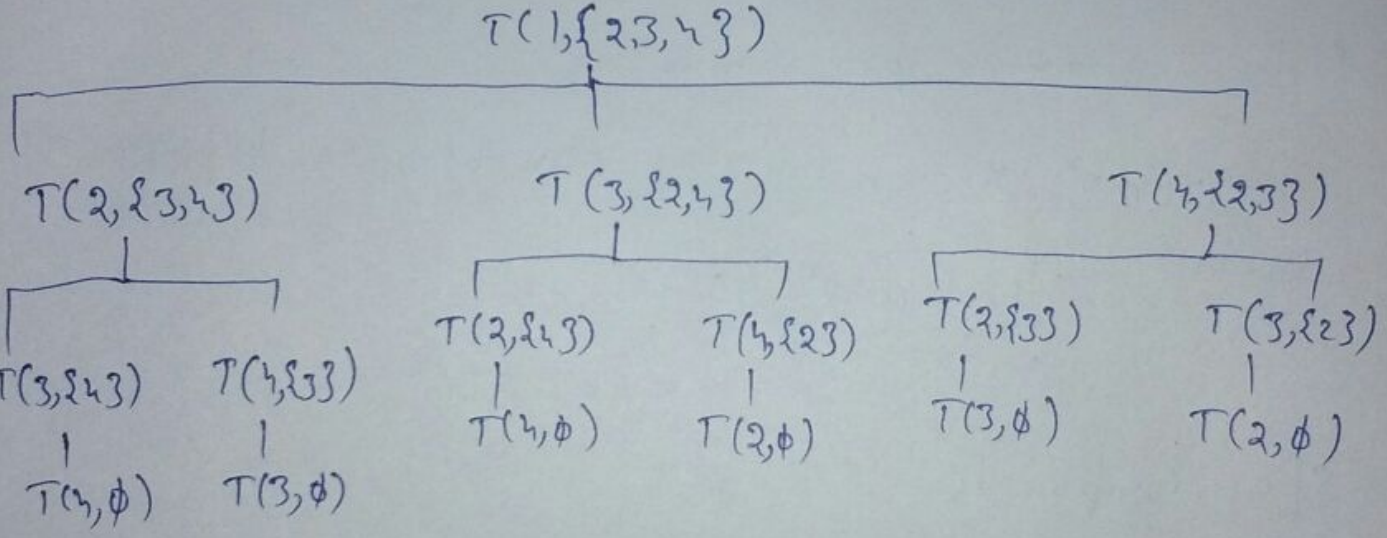


Now $T(2,\phi) = (2,1)$ | $T(4,\phi) = (4,1)$ | $T(3,\phi) = (3,1)$

Bottom up Dynamic Programming in TSP

Recursive Eqn :-

$$T(i,S) = \begin{cases} \min_{j \in S} \left((i,j) + T(j, S-\{j\}) \right) ; & S \neq \phi \\ \not{\infty}(i,1) ; & S = \phi \end{cases}$$



** Total no of unique sub problem

$$\boxed{(n-1)\,2^{n-2}}$$

Ex :- $n = 4$

$$(4-1)2^{4-2} = 3 \times 2^2 = 12$$

Complexity

Unique problem $= (n-1)\,2^{n-2}$

$= O(n2^n) < n!$

This is batter than $n!$

* In Dynamic programming

$O(n\,2^n) \times O(n)$ → Time taken to solve each problem

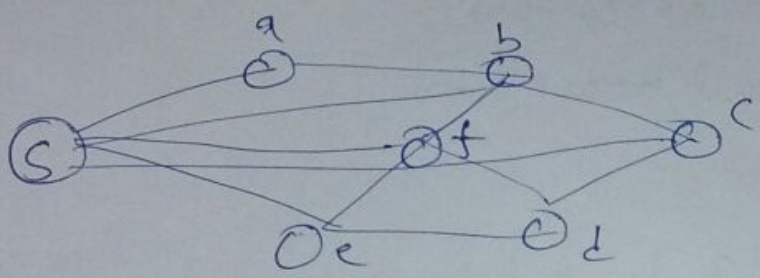
↙ Table size

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

Space complexity $= O(n2^n)$

Floyd Warshall

* Relation b/w single source path & all pair shortest



* Here we ~~find~~ in all pair shortest distance we find ~~b/w~~ distance (shorest) b/w source to each node & node to node

Ex:- (s,a) (s,b), (s,c), (s,d), (s,e), (s,f)
(a,b) (b,c), (c,d), (b,f), (f,c) - - - - - -

* ~~But in single source~~
* Complexity

If we Apply Dijkstra

$O(E \log V)$ → for dijkstra

$O(V)$ → for each ~~node~~ vertex shortest path

$O(VE \log V)$ → Complete complexity

*** we know $E = V^2$

So Complexity = $O(V^3 \log V)$ → When we apply Dijkstra

Complexity when we apply bellman ford algo for all pair shortest

$O(VE)$ → Bellman complexity

$O(V)$ → for all ~~node~~ vertex

$O(V^2E)$

$\therefore$ $\because [E = V^2]$

$O(V^4)$ = when apply bellman ford

#optimal substructure (floyed warshall)

$V = \{1, 2, 3 \text{ --- } n\}$

$i, j \in V$

$k \rightarrow$ vertex b/w i & j

$d_{ij} \rightarrow$ shortest distance

$d_{ij} =$ "p" $\rightarrow$ shortest path.

$d_{ij} =$ "p" $\rightarrow d_{ij}^{(k-1)} \rightarrow$ shortest distance without including k.

$\rightarrow (d_{ij}^{(k-1)} + d_{kj}^{(k-1)}) \rightarrow$ distance using (including k)

Floyd - Warshall Algo

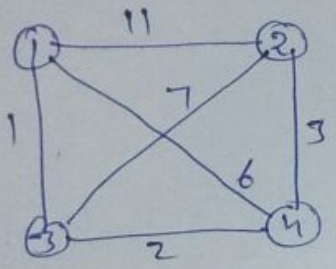
```
FLOYD-WARSHALL(W)
{
1) n = W.rows
2) D^0 = W
3) for k = 1 to n
4)     let D^(k) = (d_ij^k) be a new n x n matrix
5)     for i = 1 to n
6)         for j = 1 to n
7)             d_ij^(k) = min(d_ij^(k-1), d_ij^(k-1) + d_kj^(k-1))
8) return D^(n)
```

How Dynamic Programming Apply (in floyed Alg)

Ex:

* Here graph can be directed
* Here can be -ve edge in graph (-ve weight edge)
* No -ve edge cycle allow

Ex:



Step1

D^0 → ~~Distance indicate~~

D^0 → Distance of path contain atmost ~~only~~ one edge (only one)

$$D^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 11 & 1 & 6 \\ 11 & 0 & 7 & 3 \\ 1 & 7 & 0 & 2 \\ 6 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

D' → Distance b/w shortest path which allow go through 1 if there is a batter path.

$D' =$

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

→ (2,3) → 7 → without including 1

(2,3) ⇒ 2 → 1 + 1 → 3
11 + 1 = 12 → including 1 but it incr. cost so we are batter without using 1 so value will same 7.

→ (2,4) → 3

(2,4) → 2 → 1 + 1 → 4
11 6 → 17 ×

→ (3,4) → 2

(3,4) → 3 → 1 + 1 → 4
1 + 6 → 7 ×

D^2= Here we include {1,2} path & see just previous matrix that is $\underline{D^1}$.

$$D^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 11 & 1 & 6 \\ 11 & 0 & 7 & 3 \\ 1 & 7 & 0 & 2 \\ 6 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

→ (1,3) → 1

(1,3) → 1→2 + 2→3
11 + 7 = 18 ×

→ (1,4) → 6

(1,4) → 1→2 + 2→4
11 + 3 = 14 ×

→ (3,1) → 1

(3,1) → 3→2 + 2→1
7 + 11 = 18 ×

→ (3,4) → 2

(3,4) → 3→2 + 2→4
7 + 3 = 10 ×

→ (4,1) → 6

(4,1) → 4→2 + 2→1
3 + 11 = 14 ×

→ (4,3) → 2

(4,3) → 4→2 + 2→3
3 + 7 = 10 ×

D^3 = It will contain path {1,2,3} I see the matrix D^2

$$D^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 8 & 1 & 3 \\ 8 & 0 & 7 & 3 \\ 1 & 7 & 0 & 2 \\ 3 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

→ (1,2) → 11

(1,2) → 1→3 + 3→2
1 + 7 = 8 ✓

→ (1,4) → 6

(1,4) → 1→3 + 3→4
1 2 = 3 ✓

→ (2,4) → 3

(2,4) → 2→3 + 3→4
7 + 2 = 9 ×

→ (3,4) → 2 ✓

⇒ D^4 = It will contain {1,2,3,4}

$$D^4 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 6 & 1 & 3 \\ 6 & 0 & 5 & 3 \\ 1 & 5 & 0 & 2 \\ 3 & 3 & 2 & 0 \end{bmatrix} \end{array}$$ ✓✓

→ (1,2) = 8

(1,2) = 1→4 + 4→2
3 3→6 ✓

→ (1,3) = 1

(1,3) = 1→4→4→3
3 + 2 = 5 ×

→ (2,3) = 7 {2→4→4→3
3 + 2 = 5 ✓

Time Complexity Analysis!-

There are (n^2) matrix element or value.

& to solve we use n matrix

So of n $n(n^2) = \underline{n^3}$ → total subproblem

* work done to solve each subproblem is constant

So total time complexity =

$$O(n^3)(c) = \boxed{O(n^3)} \simeq$$

Space Complexity

$$O(n^2) \simeq$$

* Two matrix used to compute values.

NP Complete & NP Hard

Problems

- Decidable (algo exist)
  - Tractable
    If these exist at least one polynomial bound algorithm
    $O(n^k)$
  - If a problem is not tractable then it is intractable.
    $O(n^{\log n})$ $O(c^n)$
    Ex:- (Travelling Shillyman Problem)
- UnDecidable (algo not exist)
  - like Halting problem in Tms.
    (Turing Machine)

* optimisation & decision problem

optimisation Problem	Decision Problem
TSP (travelling salesman problem): A Graph G, shortest path covering all vertices exactly once.	Is there any SP covering all vertices of length atmost "k". (Yes/No)
0/1 KS: Given the capacity, profit, weights, find out the maxi. profit.	Is there any soln whose profit is atleast "k".
LCS (longest common subsequence): A, B, find LCS. eg: AA BD, AB CD → AB	Is there any ss whose length is atleast "k".

* we use only decision Problems in NP completeness theory

opt → DP (easy compare to optimisation)

(TSP) "k" → Yes / No

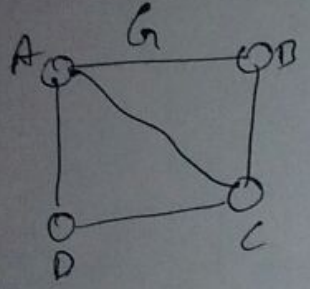
* if opt. is easy ($O(n^k)$) then DP is easy.

* If DP is hard, then opt is hard.

* verification algo

DP:

Q.) is this graph hamil?



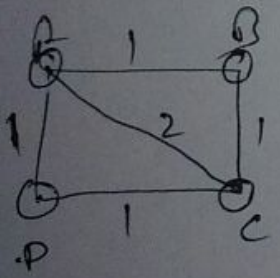
→ A – B – C – D – A

verify this ans is write or not

{ 1) check all vertex are present or not
  2) check all edges are present or not.

↓ that mins its hamil graph.

Yes Ans.



TSP- atmost 5

A – B – C – D – A

Yes =

P & NP

P:- Set of all decision problems, polynomial time algorithm to solve them.

NP:- Set of all decision problems for which there is a polynomial time verification.

P

1) for a graph G, find MST almost 10.

→ Here for a graph G we can find MST in polynomial time. Then definetly we will find MST of almost 10 value in polynomial time.

) foraction KS, profit atleast 10.

→ Here if we can solve KS problem in polynomial time then we can definetly solve KS profit atleast 10 in polynomial time

NP

if a problem P then it is definetly NP.

→ because for a problem 'P' we find solution in polynomial time then definetly we can verify its ans. in polynomial time.

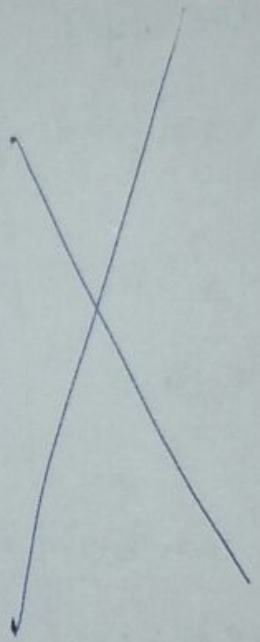
→ There are some problem which is in NP but not in P

Ex:- TSP, 0/1 Knapsack.

* P NP Introduction

P: set of all polynomial time algorithm to solve

NP: set of all decision problems for which there is a polynomial time verification.



P

→ G, MST atmost 10

eg → Prims & Kruscals

→ KS (fractional), profit is atleast 10

yes.

NP

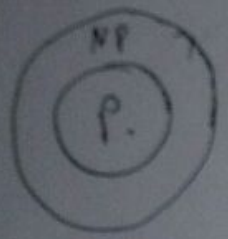
Every P Problem is in NP.

But some extra problem like TSP.

↳ Travelling salesman problem:

0/1 Knapsack

* Polynomial time reduction algo.



P - quickly solved
or
easily solved

NP :- quickly verified
or
easily varified

Poly. time Reductions :-

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 polynomical time

ii) Answer to 'α' is 'yes' if & only if answer to 'β' is 'yes'.

A $\xrightarrow[\text{'f'}]{O(n^2)}$ B $\rightarrow O(n^2)$ poly. time
α β

$O(n^k) + O(n^2)$

* if B is easy, then A is also easy
* if 'B' is in 'P', then A is also in 'P'.

A → B
| O(n^k)

* if 'A' is not in 'P' then 'B' is not in 'P'.

Ex

A: Given 'n' boolean variables with values $x_1, x_2 -- x_n$, does at least one variable have the value "True"?

B: Given 'n' integers $i_1, i_2 -- i_n$ is $\max(i_1, i_2 -- i_n) > 0$

Sln

n=4

(T, F, F, T) — T

(-30, +190, 2) → T

is there any value greater than 0.

A → B

(T, F, F, T)] A → B → O(n)

(1 0 0 1)]

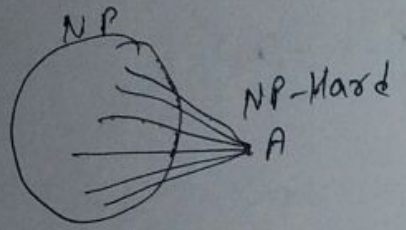
A → B O(n)

O(n)

* NP hard & NP Complete problem

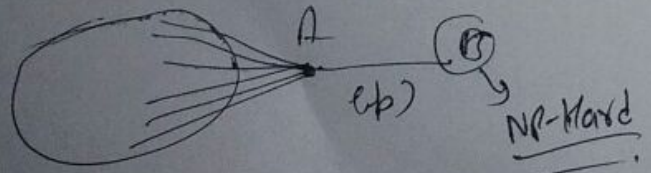
NP-hard :-

if every problem in NP can be polynomial time reducible to a problem 'A' then 'A' is called NP-Hard.



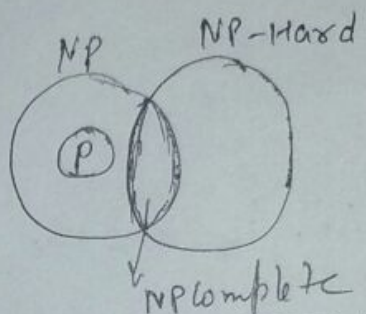
If 'A' could be solved in polynomial time, then every problem in NP is 'P'.

(P = NP) → But No one did it till now



NP Complete :

A problem is said to be NP Complete if it is 'NP' & 'NP-hard'.



→ 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 P ≠ NP.

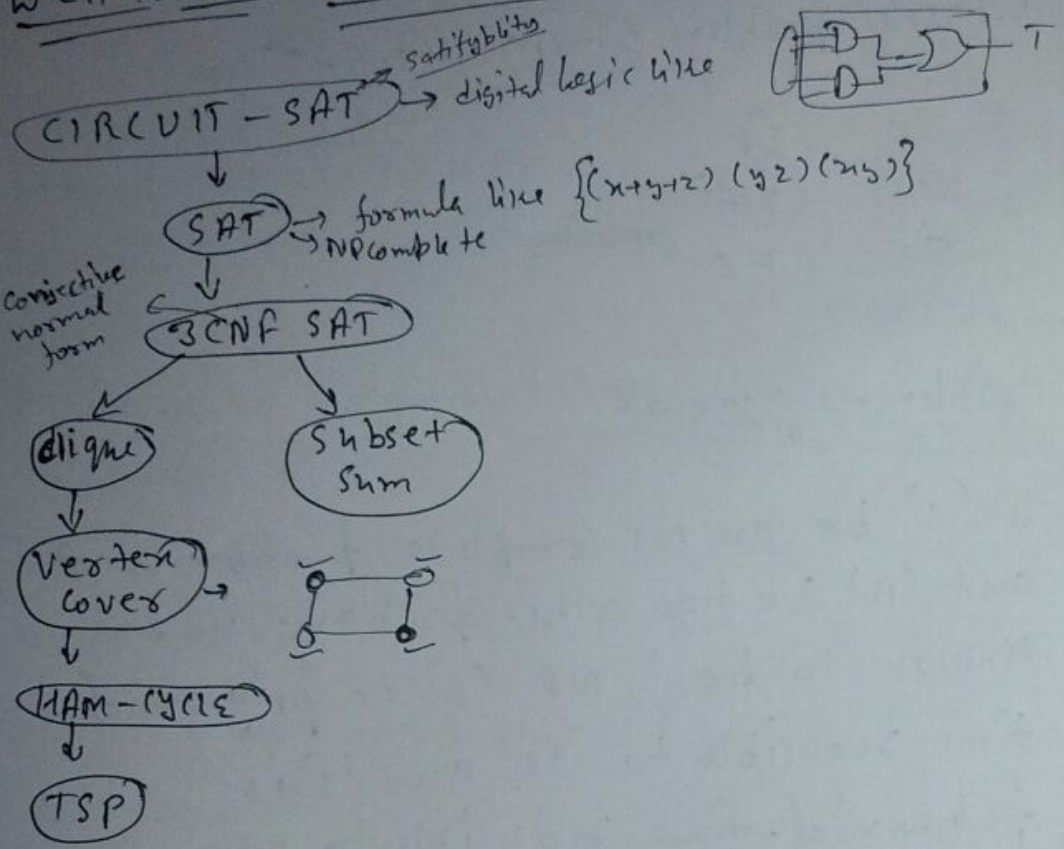
→ Status of NP is unknown.

→ If 'A' is NP-Hard & $A \xrightarrow{(\phi)} B$, then 'B' is NP-hard.

→ ~~AND if 'B' is already~~

→ If A is NP Hard & B is NP, & $A \xrightarrow{(\phi)} B$ the B is NP Complete.

* Well known NP Complete Problem



Problem-1 GATE→4

Q.) RAm & Shyam have been asked to show that a certain problem π is NP Complete RAm shows a polynomial time reduction from 3-SAT problem to π, & Shyam shows a polynomial time reduction from π to 3-SAT. What is π.

NPC (f) NP-H

3SAT $\xrightarrow{f}$ (π)

$\pi \xrightarrow{f'}$ 3SAT

Ans → π is NP Hard

Problem-2 GATE→4

NP-Complete ✓

Q.) The problem (3-SAT) & (2-SAT) are.

Polynomial

Shortest Path(P) → SP ✓ LP ✗

3SAT — NP

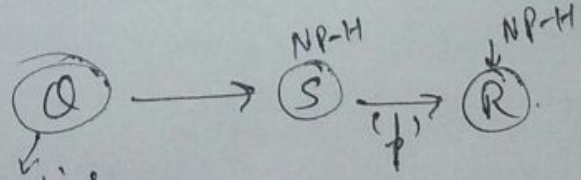
2SAT — P

EC — HC

Euler ckt (P)

Problem→3 GATE→6

Let 'S' be an NP Complete problem, & 'Q' and 'R' be two other problems not known to be in NP. 'Q' is polynomial time reducible to 'S' and 'S' is polynomial-time reducible to 'R'.



Nothing Can be Said about this.

Problem → 4 GATE-97

Q.) Let 'A' be a problem that belongs to the class NP. Then which one of the following True?

a) There is no polynomial time algo for 'A'.

b) If 'A' can be solved deterministically (all programs are) in polynomial time, then, P = NP.

c) If 'A' is NP-hard, then it is NP-complete

d) A may be undecidable.

Problem → 5

Q) Which of the following is true -

1) The problem of determining whether there exists a cycle in an undirected graph is in 'P'.

2) The problem of determining whether exists a cycle in an undirected graph is 'NP'.

3) If a problem A is NP-complete there exists a non deterministic polynomial time algo. to solve 'A'.

4) All are true.

Problem-6 GATE-92

Q). Which of the following is not NP-hard?

a) Hamiltonian circuit problem

b) 0/1 Knapsack problem.

c) Finding bi-connected components of a graph.

d) The graph colouring problem.

Q) Let SHAM$_3$ be a problem of finding Hamiltonian cycle in a graph $G = (V, E)$ with $|V|$ divisible by 3 & DHAM$_3$ be a problem of determining if Hamiltonian cycle exists in ~~such~~ graphs. Which of the following is true?

HC – NPC

DHAM → NPC

SHAM → $|V| \equiv 0 \bmod 3$

DHAM $\xrightarrow{\propto}$ SHAM

SHAM → NP-H

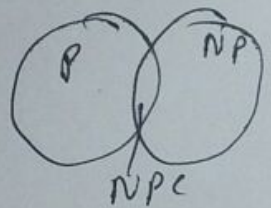
SHAM → NP

Prob → 8

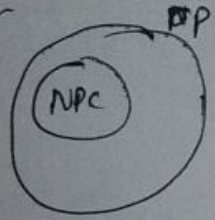
Q. Suppose a polynomial time algorithm is discoverd that correctly computes the largest clique in a give graph. In this scenario, which one of the following represents the correct venn diagram of the complexity classes P, NP & NPC ?

a)

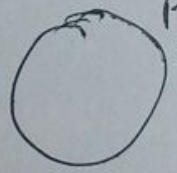
b)



c) ✓



d)



Prob-9

a) Consider the decision problem 2CNF-SAT

a) NP Complete

b) Solvable in polynomial time by reduction to directed graph reachability.

c) solvable in constant time since any input instance is satisfiable.

d) NP-Hard, but not NPC.

Problem-10

If an NP-hard problem can be solved in polynomial time, then P=NP (T/F)

Problem-11

The set of NP problem is not closed under (polynomial time) (reductions) (T/F)

↳ it's a unary operation