17 January 2022 09:51

Sem 2 → Alphabets

Sem 3 → Tree → BST, AVL, B, B+ } Trees

Graph → Adjacency matrix, Adjacency list → DFS → Stack, BFS → Queue

Sem 4 → Analysis of Algorithm.

Algorithms → It is finite set of Instruction.

Features →

(1) Input → Zero or more external quantities supplied.

(2) Output → Atleast one output

(3) Definite → Instructions must be clear and unambiguous.

Algo: To perform Binary Search.

Input: An array of n element, and X to Search.

O/P → 1 or 0 (Mila) (Nahi)

Prereq → It should be sorted → sort(a) } Ambiguity

(4) Finite → Algo must terminate after finite number of steps.

(5) Effective → Must satisfy the goal.

Analyze the Algorithm

Complexity (Order of growth)

- Time Complexity = T(P) → Time req to solve given problem P for input size n.
  - In Analysis Part → T.C ∝ No of comparison in Algo.
  - → Processor Time.
  - → Understanding → Order of Growth
- Space Complexity → Amt of Primary memory (RAM) needed to solve a problem of size n
  - → We also express it in terms of order of growth.

Vimp

Order of growth → If input (n time) se increase hoga to Time required kitne time se increase hoga.

Order of growth = n^2 ✓

Time req

If (10 time) $n = 100$ → 100 ms. (100 time)

$n = 1000$ → $100 \times 100 = 10000$ ms

Agar n is increased by 10 time

Input	Time Req	
n = 100	100 ms	Order of growth ⇒
n = 1000	7000 ms (10 time)	

No of Comparison.

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

n = 5 → 5 Comp.

n = 10 → 10 Comp.

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

⇒ Time Complexity

Order of growth = n ✓

n = 5 → 5 comp.
n = 10 → 10 comp.
n = 15 → 15 comp.
n = 100 → 100 comp.

10 time. (10 time)

Time — Agzent — SSD — less time | i3 — HDD — more time

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

n = 10

for each outer iteration
inner ⇒ 10 time.
There are 10 outer iteration

No of comp → 100 comp. → 100 time

∴ n = 10 → 10 time

n = 100 → Outer × Inner ⇒ 1000 comp.
100 100

T.C of above loop = n^2

* Technically Time Req to Solve problem — Depends → Hardware

* Conceptually just to make it Independent of H/W
we measure in terms of ⇒ No of Comparisons

Consider

```
for (i=1; i<=100; i=i+2)
```

```
for (i=1; i<=100; i=i+2)
{
```

for(i=1; i<=100; i=i+2)

{ => .

}

i = i+1 ⇒

i = i+2

n=100 → 100 comps

n=100 ⇒ 50 comp.

n=200 ⇒ 100 comp

n=300 ⇒ 150 comps.

n=500 ⇒ 250 comps

5 time

5 time

= O(n)

{

}

n=100

n=64

100

i=1
i=2
i=4
i=8
i=16
i=32
i=64

6

$\log_2 64 = \log_a b$

$= 6 \log_a a$

$= 6$

O(log n)

Space Complexity → Amt of Space (RAM) required by the Algorithm wrt to input to give result.

$S(P) = C + S_p$.

$S(P)$ ↑ Space req to solve problem P

C = constant part ⇒ It does not depend on input (Instruction Space, constant values)

S_p ⇒ Variable part ⇒ Depends on input → Depends on Space needed by Variable Component where size depends on program instance

Consider

```
int n;   → 2 byte
int *a;  → 8 byte
```

~~int a[10];~~ static

```
printf("Enter value of n\n");
scanf("%d", &n);
a = (int *) malloc (sizeof(int));
```

Array of size n.

Constant part = n (2 byte), p (8 byte) RAM

Size/space needed by n & pointer p is fixed, it does not depend on value of n.

Space ↗ Constant part ↘ Variable part = Size of array ⇒ it depends on value of n → Allocation is done Runtime ✓

if n=5 → 5×2 → 10 byte

if n=100 → 100×2 → new byte

10 time

if n=1000 → 1000×2 => 2000 byte

10 time

If n = 1000

1000 × 2 => 2000 byte

Order of Growth of Space Complexity = n

Consider . Aaj kuch bhi bol Aur Hoga ->

```
for (i=1; i<=n; i++)
{
    for (j=i; j<=n; j++)
    {
        for (k=1; k<=n; k++)
        {
            printf("Hello\n");
        }
    }
}
```

$$\sum_{i=1}^{n} \sum_{j=i}^{n} \sum_{k=1}^{n} 1 \quad [1+1+1 \ldots\ldots n]$$

$$\Rightarrow \sum_{i=1}^{n} \left(\sum_{j=i}^{n} n \right)$$

$$= \left[n \sum_{j=i}^{n} \right] \quad i, i+1, i+2, \ldots\ldots n-1, n$$

$a = i$, $d = 1$

$T_n = a + (n-1)d$

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

$= \frac{n}{2}\left[a_1 + a_n\right]$

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

$$= \sum_{i=1}^{n} ni + \frac{n^2}{2} - \frac{n}{2}$$

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

n^3

10 time

1000 time.

=

24 January 2022 08:53

Time Complexity —Represents→ Order of Growth.

→ [Agar input ko n time se increase kiya toh problem solve karne ke liye jo time lagega woh kitne time se increase hoga].

∝ No of Comparison in the Algorithm.

Analysis → Type of Analysis.

A Posteriori Analysis

1. Here analysis is done only after the Execution on machine.
2. It depends on Programming language and Hardware of the System.
3. It will give Exact answer (based on real time)
4. The Answer can change from System to System.
5. It is relative Analysis
6. It does not use Asymptotic Notations.

A Priori Analysis

1. Analysis done before the Execution of Algo in System.
2. It is Independent of language and hardware of the System.
3. It will give approximate Answer (based on Assumption).
4. Answer remains same on all the System.
5. It is absolute analysis
6. It uses Asymptotic Notations

[O → Big O Notation
Ω ⇒ Omega Notation
Θ ⇒ Theta Notation]

Time Complexity → $T(p)$ ⇒ Time req for an Algorithm to give o/p. wrt given input size.

⇒ gt is Processor Time.

Time Complexity = Compile Time + Run Time.
(constant) (input dependent) (variable)

↓ gt does not depend on input characteristic

↓ gt depends on input characteristic

Run Time depends on input characteristic (t_p)

$$T(p) = C_p + t_p$$

C_p: Compile Time (constant)
t_p: Runtime.

```
int main ()
{  int n, *p, i;
   printf("Enter value of n\n");
   scanf("%d", &n);
   p = (int *) malloc (sizeof(int));
   printf("Enter elements in array\n");
   for (i=1; i<=n; i++)
   scanf("%d", (p+i));
OR scanf("%d", &p[i]);
}
```

Here Compile Time = constant
= does not change with n.

Run Time = Variable Component
= varies with n.

24 January 2022 09:53

How to Represent/Express Time Complexity of Algorithm →

Asymptotic Notations →

* These Notations denotes Asymptotic Analysis of Algorithms.

* Asymptotic Analysis ⇒ Analysis performed for very large value of n is known as Asymptotic Analysis.

* To analyze the efficiency of an Algorithm it is not necessary to conduct detailed analysis of Running time, but we emphasize more on finding "how Running Time increases with increase in input" (This is known as Order of Growth)

Uddeshya !!

"Jab main input ka size 'x' time se increase karoonga toh Execution time kitne time se increase hoga"

Ex → If input size is 100 and time req → 0.5ms
then if input size is 500 → ?

Question Arises → What is Order of Growth [No of steps]

(Case 1) Let say → n^2 [Quadratic Order of Growth].

then $n^2 \rightarrow (100)^2$ steps

(case 1) let say [illegible]

then

input size = 100 ⟶ No of steps = $n^2 \Rightarrow (100)^2$ steps

for $(100)^2$ steps ⟶ 0.5ms

input size = 500 ⟶ No of steps = $(500)^2$.

for $(100)^2$ steps ⟶ 0.5ms

$(500)^2$ steps ⟶ x.

$$x = \frac{(500)^2 * 0.5ms}{(100)^2} = 12.5ms.$$

(case 2) let order of growth = n. [Linear Order of Growth]

for n = 100 ⟶ No of steps 100 ⟶ Time req 0.5ms.

for n = 500 ⟶ 500 ⟶ x.

$$x = \frac{500 \times 0.5}{100} = 2.5ms.$$

When we want to Compare 2 Algorithms wrt the large input size, A useful methodology called "Order of growth" is used

In Asymptotic Analysis, it is considered that an Algorithm(A_1) is better than better than A_2 if Order of Growth of Running Time of A_1 is better (lower) than that of A_2. Algorithm.

∴ if order of growth of A_1 < Order of growth of A_2 then Algorithm A_1 is Better than A_2.

Let say Order of growth of A_1 $\Rightarrow n$. [Linear]
and order of growth of A_2 $\Rightarrow n^2$ [quadratic]

∴ A_1 is Better than A_2.

* Order of growth can be estimated by taking into account the dominant term of Expression.

Consider. Exp is $n^2 - 2n + 5$ ≈ n^2

for very large value of n (Let say 100000) $\Rightarrow (100000)^2 - (2*100000) + 5$

≈ $(100000)^2$.

∴ for the Exp $n^2 - 2n + 5$ $\Longrightarrow$ Order of Growth = n^2.

VVImp Explain Asymptotic Notations.

* These are Notations used to express the time Complexity of Algorithm.

Time Complexity

- Best Case → Lower Bound → Kam se kam Kitna Time lagega for given input
  Notation → Omega Ω
- Worst Case ⇒ Upper Bound → Jada se jada Kitna time lagega for given Input.
  Notation → Big O ⇒ O
- Average Case → Average Case Complexity → When we have both upper & lower Bound.
  Notation Theta = Θ

Consider Bubble Sort →

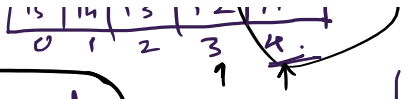
```
for(i=0;i<n-1;i++)          // No of comparison.
{
  for(j=0; j<n-1; j++)
  {
    if(a[j] > a[j+1])
    {
      t=a[j];               // Assume time req for
      a[j]=a[j+1];          // Swapping is negligible
      a[j+1]=t;
    }
  }
}
```

Consider Input: n=5

15	14	13	12	11
0	1	2	3	4

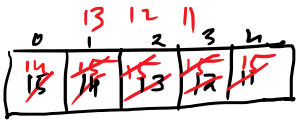
i=0 | i=1 | i=2 | i=3

Consider Input.



This could be Worst Input ⇒ $i=0$, $j=0,1,2,3$ | $i=1$, $j=0-3$ | $i=2$, $j=0-3$ | $i=3$, $j=0-3$

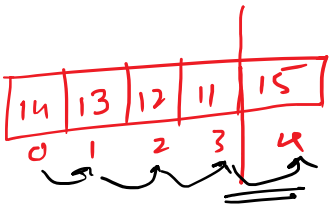
1st Iteration



$[n=5]$

When $i=0$ → 4 comp.
$i=1$ → 4 comp
$i=2$ ⇒ 4 comp.
$i=3$ ⇒ 4 comp.
4 iterations.

$i=0$
$j=0$ $a[0]>a[1]$ Yes - swap
$j=1$ $a[1]>a[2]$ Yes - swap
$j=2$ $a[2]>a[3]$ Yes - swap
$j=3$ $a[3]>a[4]$ Yes - swap.



4 comparisons.

for n
$i=1$ ⟶ $(n-1)$ comp.
$i=2$ ⟶ $(n-1)$ comp

$i=n-1$ ⟶ $(n-1)$ comp.
$(n-1)$ iterations

Total no of comp = $(n-1)+(n-1)\cdots$ $(n-1)$ time.

$= (n-1)*(n-1) = n^2-2n+1$

Order of Growth $= n^2$

For Worst Case Input ⟶ Order of Growth = n^2

$= O(n^2)$

Case 2> Consider Modified Bubble Sort

```
for(i=n-2; i>=0; i--)
    for(j=0; j<=i; j++)
        if(a[j]>a[j+1])
            swap;
```

$n=5$
$i=3$
$j=$ 0→1, 1→2, 2→3, 3→4
$(n-1)$ comp
4 comp

$i=2$
$j=$ 0→1, 1→2, 2→3
$(n-2)$ comp

$i=1$
$j=$ 0→1, 1→2
$(n-3)$ comp

$i=0$

∴ Total No of Comp

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

1st Iteration, 2nd Iteration, last

$j = 0 \to 1$

1 comparison.

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

Case 3) Consider $n = 5$

11	12	13	14	15
0	1	2	3	4

Consider Sorted Input Given →

Comp $n = 5$.

$i = 0$

$j = 0\ 1$, $1\ 2$, $2\ 3$, $3\ 4$ } No Swapping.

4 Comp.

$(n-1)$

$i = 1$

$j = 0-1$, $1\ 2$, $2\ 3$, $3\ 4$

4 comparison. (No Swapping)

$i = 2$.

4 Comp (No Swapping)

$i = 3$

4 Comp (No Swapping)

No of Comp $= (n-1) * (n-1) = n^2 - 2n + 1 = n^2$

∴ Best Case Time Complexity $= \Omega(n^2)$.

∴ For Bubble Sort

Best Case T.C ——— $\Omega(n^2)$

Worst Case T.C ——— $O(n^2)$

Avg Case TC ⟶ $\Theta(n^2)$

Big O Notation →

↳ It is used to represent/express Upper Bound of Algorithm

↳ It represents Worst Case Time Complexity of Algorithm.

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

i.e $f(n)$ is Big O of $g(n)$.

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

where c is +ve constant > 0 and $n \geq n_0$ where $n_0 \geq 1$

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

this holds. ⇒ $f(n) \leq c \cdot g(n) \quad \forall\, n \geq n_0$ where $n_0 \geq 1$ and $c > 0$

if this is satisfied

$g(n)$ is upper Bound of $f(n)$

(1) We say $f(n) = 3n+2$ $\quad g(n) = n$, let $c = 4$

$3n + 2 \leq 4n$ this should be true for $\forall n$

$n=1$ $\quad 3(1)+2 \leq 4(1)$ false

$n=2$ $\quad 3(2)+2 \leq 4(2)$ True.

$n=3$ $\quad 3(3)+2 \leq 4(3)$ True

True for $\forall n \geq 2$ $\quad n_0 = 2$

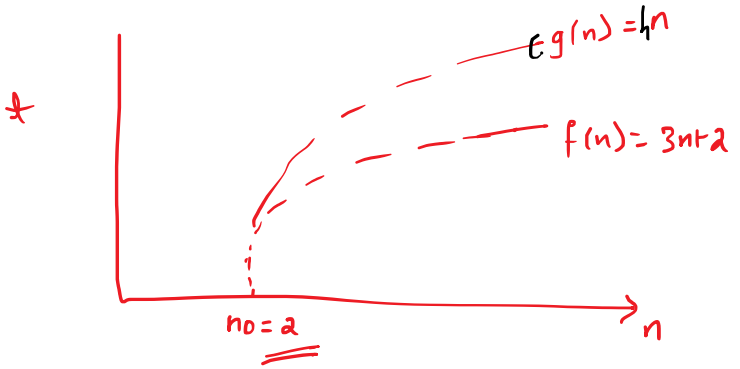
$n=2$

$n=3$ $\quad 3(3)+2 \leq 4(3)$ True

$n=4$ $\quad 3(4)+2 \leq 4(4)$ True

True for $\forall n \geq 2$ $\quad \therefore n_0 = 2$

$f(n) = Og(n) \quad \forall n \geq 2, \; c = 4$ Satisfied.



Consider $f(n) = 3n+2, \quad g(n) = n$ let $c = 6$

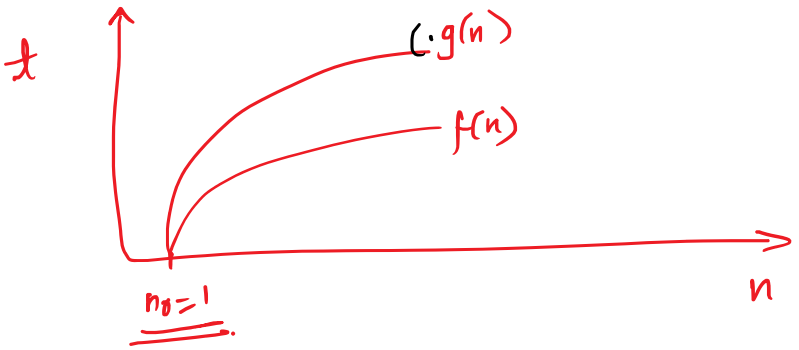
$n=1$ $\quad 3n+2 \leq 6n$

$3(1)+2 \leq 6(1)$ True

$3n+2 \leq 6n \; \forall n \geq 1$

Here $n_0 = 1$

$\therefore$ Here $f(n) = O(g(n)) \; \forall n \geq 1, \; c = 6$



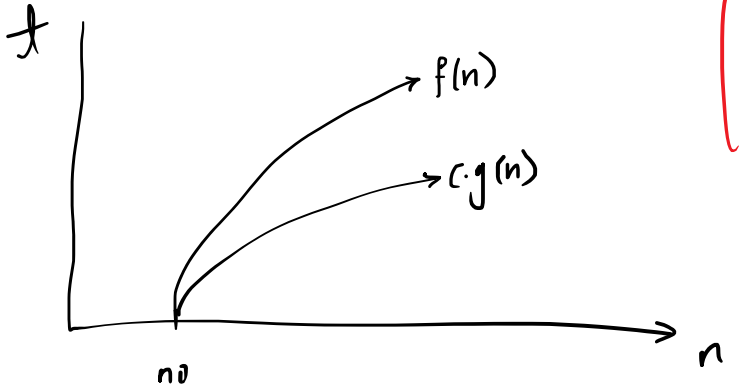
Omega Notation (Ω)

→ It is used to express/represent Best case/Lower Bound of Algorithm.

→ $f(n) = \Omega\, g(n) \Longrightarrow$ $g(n)$ is lower bound of $f(n)$.

$f(n) \geq c \cdot g(n)$ $\forall\, n \geq n_0$ where $c > 0$ & $n_0 \geq 1$

[for this particular c the cond^n $f(n) \geq c \cdot g(n)$ must be true $\forall\, n \geq n_0$]



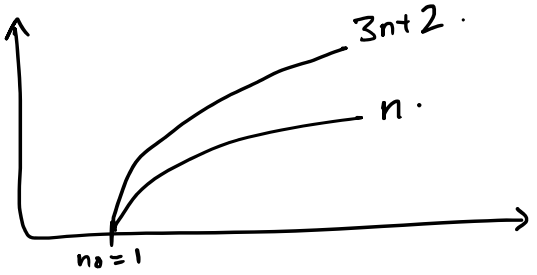
Consider $f(n) = 3n+2$ $g(n) = n$. let $c = 1$

$3n+2 \geq 1 \cdot n$

$n=1$: $3(1)+2 \geq 1$ → True

$n=2$: $3(2)+2 \geq 2$ True

} Here $n_0 = 1$



(2) Let $f(n) = 3n+2$ $g(n) = n$ let $c = 6$

(a) $3n+2 \geq 6n$ this will not hold True.

$n=1$ $5 \geq 6$ False.

$n=2$ $8 \geq 12$ False

$n=3$ $11 \geq 18$ False

Average Case Complexity (Theta Θ) Notation:

Consider $f(n) = \Theta\, g(n)$. if there exists +ve constants c_1 and c_2 and n_0 such that

$c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n)$ $\forall\, n \ge n_0$ where $c_1, c_2 > 0$ and $n_0 \ge 1$.

$f(n) = \Omega g(n)$ (Best Case).

$f(n) = O g(n)$. (worst Case).

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

Consider $f(n) = 3n+2$

let $g(n) = n$.

Now

"it is true for $n_0 = 1$, just to have common n_0 we keep it 2."

$3n+2 \ge 3n$ where $c = 3$, $n_0 = 2$

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

$3n+2 \le 4n$ where $c = 4$, $n_0 = 2$

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

$3n \le 3n+2 \le 4n$ for $n_0 \ge 2$

($3n$: c_1; $4n$: c_2)

$\therefore$ $3n+2 = \Theta(n)$ for $c_1 = 3$, $c_2 = 4$ and $\forall\, n \ge n_0$ where $n_0 = 2$.

$f(n)$ $\quad$ $g(n)$

$\Rightarrow$ $f(n) = \Theta\, g(n)$ for $c_1 = 3$, $c_2 = 4$ & $\forall\, n \ge n_0$ where $n_0 = 2$

07) find the time required when Algo takes 0.5 ms for input size = 100. How long will it take for input size 500 when the complexity is

(a) linear (b) Quadratic (c) nlogn [common log base 10] (d) cubic

Note → Complexity ⇒ Order of growth ⇒ no of steps for input n.

(a) linear order of growth.

if n = 100 ——→ No of steps = 100

if n = 500 ——→ No of steps = 500

for n = 100 i.e 100 steps time ⇒ 0.5 ms

for n = 500 i.e 500 steps time ⇒ x.

$$x = \frac{500 \times 0.5}{100} = 2.5 \text{ ms}$$

(b) Order of growth = quadratic.

if n = 100 ——→ No of steps = $(100)^2$

if n = 500 ——→ No of steps = $(500)^2$

for n = 100 i.e $(100)^2$ steps — 0.5 ms

for n = 500 i.e $(500)^2$ steps — x.

$$x = 12.5 \text{ msec}$$

(c) Order of growth = $n \log n$.

if $n = 100$ ⟶ No of steps = $(100) \log 100$ steps ⟶ 0.5 ms.

if $n = 500$ ⟶ no of steps = $(500) \log 500$ steps ⟶ x.

$$x = \frac{0.5 * 500 \log 500}{100 \log 100} = 3.3737 \text{ ms}$$

(d) Order of growth = cubic.

if $n = 100$ ⟶ No of steps = $(100)^3$ steps ⇒ 0.5ms

if $n = 500$ ⟶ No of steps = $(500)^3$ steps ⇒ x.

$$x = 62.5 \text{ ms}.$$

Q Check if O, Ω, Θ is possible or not.

[If O and Ω is possible then only Θ is possible]

① $f(n) = n+5$

$g(n) = n$.

Check for 'O'.

$n+5 \leq c \cdot n$.

Now find c and n_0 to satisfy this

Let $c = 2$

$n+5 \leq 2n$

for $n_0 \geq 5$

$\therefore n+5 \leq 2n \quad \forall n \geq 5$

$c_1 = 2$

Big O is possible.

Check for Ω

$n+5 \geq c \cdot n$

Now find c & n_0 to satisfy this

Let $c = 1$

$n+5 \geq n$.

true for $\forall n \geq 1$

$\therefore \quad n+5 \geq n \quad \forall n \geq 1$

$c_2 = 1$

Ω is possible

for Θ

we have $n+5 \geq n \quad \forall n \geq 1$ —I (can be written as) $n+5 \geq n \quad \forall n \geq 5$ (I)

& $n+5 \leq 2n \quad \forall n \geq 5$ —II

$\therefore \quad n \leq n+5 \leq 2n$ true $\forall n \geq 5$

$c_1 = 1$ $\quad$ $c_2 = 2$

$\therefore \quad (n+5) = \Theta(n)$ $\quad \therefore \Theta$ is possible

Q chk if Θ, Ω, O is possible.

$f(n) = n^2 \quad g(n) = n^3$.

Let chk for O $\quad f(n) \leq c \cdot g(n)$

$n^2 \leq c \cdot n^3$

let $c=1$, $n_0=1$

$n^2 \leq n^3$

for $n_0 = 1$

$1 \leq 1$ ✓

for $n_0 = 2$

$2^2 \leq 2^3$ ✓

$\therefore$ $f(n) = Og(n)$ $\forall$ $n \geq n_0$ for $c=1$ and $n_0 \geq 1$

Big O is possible.

Let chk for Ω $\quad f(n) \geq c \cdot g(n)$.

$n^2 \geq c\, n^3$

This is possible only when $c = \frac{1}{n}, \frac{1}{n^2} \ldots\ldots$

But c has to be constant

$\therefore$ Ω is not possible

Since Ω is not possible, even Θ is not possible.

Q) Arrange the following in Increasing order.

(a) $n^{1/3}$ (b) e^n (c) $n^{7/4}$ (d) $n(\log n)^9$ (e) $(1.000001)^n$.

Note $e = 2.71828$

Sol first Lets Compare e^n and $(1.000001)^n$

$\therefore$ $(2.71828)^n > (1.000001)^n$.

$\therefore$ $e^n > (1.000001)^n$.

Now Consider $n^{1/3}$ and $n^{7/4}$

$n^{0.3} < n^{1.7}$

Now Compare $n^{1/3}$ and $n(\log n)^9$

$n^{0.3}$ $n^1(\log n)^9$

For very large value of n $n^{0.3} < n^1(\log n)^9$.

$\therefore$ $n^{1/3} < n^1(\log n)^9 < n^{7/4} < (1.00000)^n < e^n$

Let $n = 100000$ $n(\log n)^9 = (10)^5$

Q) Find the Time Complexity

1. Chk no of stmts
2. How many time a statement will Execute (order of mag)
3. Add all order of magnitude.

(1)

```
main()
{
  x=y+2;          ---- 1 time
  for(i=1 to n)
    x=y+2;        ----> n time
  for(i=1 to n)
  {
    for(j=1 to n)
    {
      x=y+2;      ----> n^2 time
    }
  }
}
```

Total = $1+n+n^2$ = Upper Bound = $O(n^2)$.

(2)

```
for(i=1 to n)
  stmt      ----> n time
for(i=1 to n)
  stmt      ----> n time
```

$\Rightarrow n+n = 2n$

Total = $O(n)$

(3)

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

$\Rightarrow$ n time

(4)

```
i=1
while(i<=n)
{
  i=i+5;
}
```

$\Rightarrow n/5$ time

n = 100
i = 1
i = 5
i = 10
i = 15
:
20 time

```
i=n;
while(i>=1)
{
  i=i-5;
}
```

$\Rightarrow n/5$ time.

If `i = i ± k` — lwp will Execute $\rightarrow$ n/k time.

(5) `for(i=1; i<=n; i=2*i)`

(5) for(i=1; i<=n; i=2*i)

stmt.

$i=1 \Rightarrow 2^0$

$i=2^1$

$i=4 \Rightarrow 2^2$

⋮

At we say lwp Executes K time

At K^{th} iteration $i=2^K=n$

In last iteration $\therefore$ At K^{th} iteration $= i = \boxed{2^K = n}$

$K \log 2 = \log n$

$\boxed{K = \log_2 n}$

lwp Executes K time $\Rightarrow$ $\boxed{\log_2 n}$ time.

(6) for(i=1; i<=n; i=i*3)

stmt.

$i=1 \Rightarrow 3^0$

$i=3 \Rightarrow 3^1$

$i=9 \Rightarrow 3^2$

let lwp Executes K time

$\therefore$ In last iteration $i=3^K=n$.

$\therefore \boxed{K = \log_3 n}$

$\boxed{\text{lwp Executes } \log_3 n \text{ time}}$

(7) for(i=n; i>=1; i=i/3)

stmt.

$i=n/3^0$

$i=n/3^1$

$i=n/3^2$

let lwp Executes K time

$\therefore$ In last iteration $\boxed{i=n/3^K=1}$

$\therefore \frac{n}{3^K} = 1$

$\therefore n = 3^K$

$\therefore \quad n = 3^K$

$$K = \log_3^n$$

$\therefore$ loop Executes $\log_3^n$ time.

*

if $i = i * K$ or $i = i / K$ then loop Executes $\log_K^n$ time.

⑧ `for( i=2; i<=n; i = i*2 )`
`stmt.`

$i = 2^{2^0}$
$i = 2^{2^1}$
$i = 2^{2^2}$
$i =$

Let there be K iteration
$\therefore$ In last iteration

$i = 2^{2^K} = n$

Take log on BS.

$2^K \log 2 = \log n$

$2^K = \log_2^n$.

Take log on BS.

$K \log 2 = \log(\log_2^n)$

$$K = \log_2(\log_2^n)$$

`for(i=n; i>=1; i = i^1/2)`
`stmt`

$i = n^{1/2^0}$
$i = n^{1/2^1}$
$i = n^{1/2^2}$

Let K iterations.
In last iteration

$$i = n^{1/2^K} = 1$$

$n^{1/2^K} = 1$

$\frac{1}{2^K} \log n = \log 1$

$\log n =$

⑨ `for( i=a^2; i<=n; i = i^3 )`
`stmt`

$i = a^{3^0}$
$i = a^{3^1}$
$i = a^{3^2}$

$\therefore$ Let K iteration
In last iteration

$$i = a^{3^K} = n$$

$i = 2^{3^1}$

$i = 2^{3^2}$

$i = 2^{3^k} = n$

$\therefore \quad 3^k \log 2 = \log n$

$3^k = \log_2 n$

$k = \log_3 (\log_2 n).$

If first value $i = a$, and $i = i^k$

loop Execute = $\log_k (\log_a n)$ //.

a = first value

k = power.

Recurrance Relation →

Consider : $T(n) = 4T(n/2) + n^2$ } cost

- $T(n)$: Time req to solve problem of size n
- $4T(n/2)$: Time req to solve problem of size $n/2$
- n^2 : overhead } Time for Recursion and Merging Soln

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

- $T(n)$: problem of size n
- $T(n/5)$: sub problem of size $n/5$
- $T(4n/5)$: subproblem of size $n/5$ to $4n/5$
- n : overhead

Approach →

Here Problem is divided into subproblem of smaller size.

Combine soln of subproblems to get the soln of Complete Problem

* Effort is Recursively Applied on subproblems of different size (smaller)

Recurrance Relation — Process Wahi, But Size Alag.

Solving Recurrance Relation :→

1. There will be Recurrance Relation given.
2. Find Time Complexity of the Recurrance Relation.

To Solve Recurrance Relation →
- Master method (M Imp)
- Substitution Method.
- Recursion Tree Method

Substitution Method.

① Given $T(n) = 1$ if $n = 1$ ⟶ Terminating condition.

$= T(n/2) + n$ if $n > 1$.

Using Substitution Method.

subproblem

$\longrightarrow T(n) = T(n/2) + n$.

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

$= \left[T\left(n/2^3\right) + \frac{n}{2^2}\right] + \frac{n}{2^1} + \frac{n}{2^0}$

At k^{th} step $= T\left(n/2^k\right) + \frac{n}{2^{k-1}} + \cdots + \frac{n}{2^3} + \frac{n}{2^2} + \frac{n}{2} + \frac{n}{2^0}$

Let at k^{th} step problem Size = 1

But at k^{th} step $= \frac{n}{2^k}$

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

$2^k = n$

$\boxed{k = \log_2 n}$

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

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

$= 1 + n\left(\frac{1}{2^0} + \frac{1}{2^1} + \cdots + \frac{1}{2^{\log_2 n - 1}}\right)$

$= 1 + n \left(\frac{a^0 \quad a^1 \quad \cdots \quad a^{\log_2 n}}{} \right)$

G.P: $a = 1$, $r = 1/2$, $n = \log_2^n + 1$, $S_n = \frac{a * (1 - r^n)}{1 - r}$

Here $r < 1$

$= 1 + n \left(\frac{1 * (1 - 0.5^{\log_2^n - 1})}{1/2} \right)$

$= 1 + 2n \left(1 - 0.5^{\log_2^n - 1} \right)$

$\approx 1 + 2n$

$\approx O(n)$.

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

$= 1 \quad n = 1$

Solve using Substitution method →

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

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

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

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

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

$= T(n-6) + 1 + 1 + 1$

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

At k^{th} step $= T(n-2*k) + (1+1+\dots k \text{ time})$

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

But here problem size = 1

$\therefore n-2k = 1$

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

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

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

$= O(n)$ linear order

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

$= 1 \quad n = 1$

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

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

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

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

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

At k^{th} step $= 2^k T(n/2^k) + kn$.

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

$\therefore \boxed{k = \log_2^n}$

$= 2^{\log_2^n} T(1) + n * \log_2^n$

$= n + n\log_2^n$.

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

$n = 10000$

$n = 100000 + 10000 * \log_2^{10000}$

$+ 100000 * 13.28$

Recursion Tree Method

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

Here in every step we have more than one Recursive call.

If Recurrence Relation has more than one Recursive call then we use Recursion Tree method.

Steps

1. Here draw Recursion Tree from given Relation.
2. Cost of Each level of Tree needs to be calculated (sum of cost of all the nodes at all the level).
3. Total cost of Tree = Sum of cost of all the nodes at all the level.

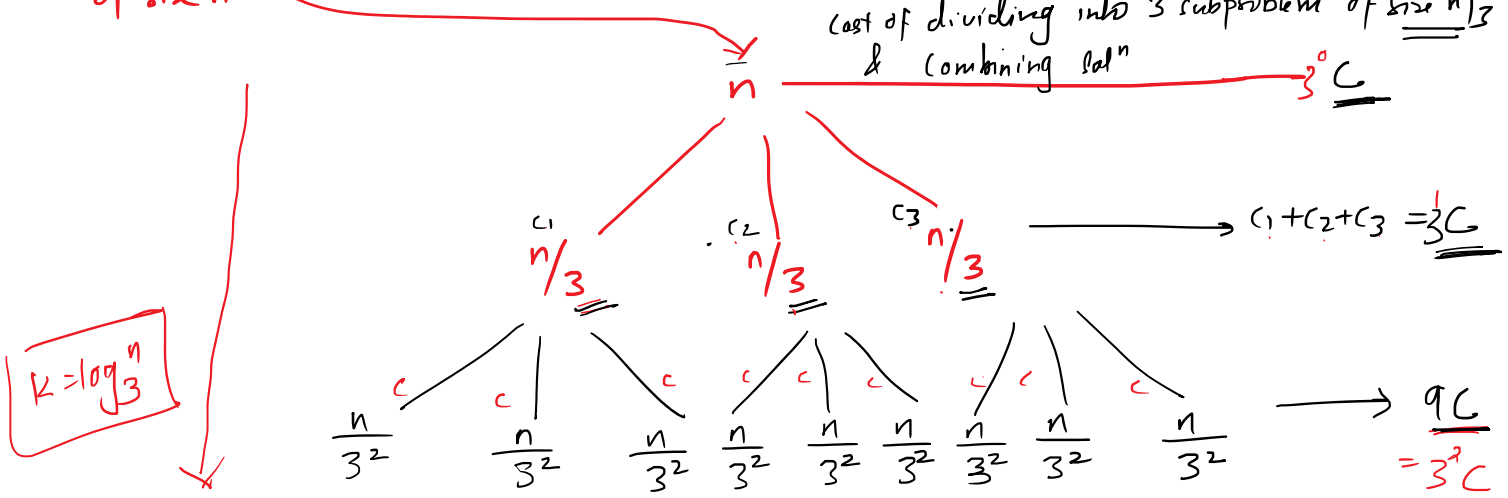
Ex $T(n) = 3T(n/3) + C$

dividing into 3 subproblems of size $n/3$

C : cost/overhead.

$T(n)$: cost of problem of size n

$3T(n/3)$: problems of size $n/3$



→ Let there be k levels at k^{th} level problem size = 1

but at k^{th} level = $\frac{n}{3^k}$ $\therefore \frac{n}{3^k} = 1$ $3^k = n$ $k = \log_3 n$

Total Cost = $3^0 C + 3^1 C + \cdots 3^{\log_3 n} C = C(3^0 + 3^1 + \cdots 3^{\log_3 n})$

Total Cost = $3^0C + 3^1C + \cdots 3^{\log_3 n}C = C(3^0 + 3^1 + \cdots 3^{\log_3 n})$

=

Binary Search

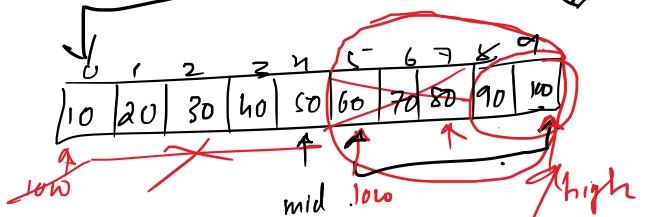
① Calculate Mid.

② If $x < a[mid] \Rightarrow high = mid - 1$

③ If $x > a[mid] = low = mid + 1$

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

mid → $\frac{l+h}{2}$ ⇒ constant time



x = 90

low = 0

high = 9

mid = $\frac{0+9}{2}$ = 4

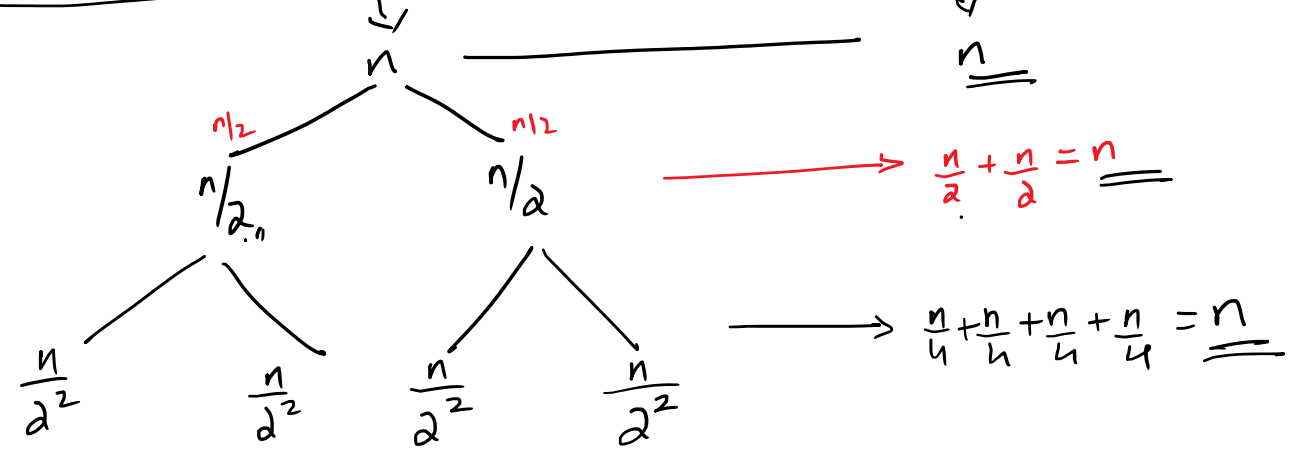
Is x == a[4] No

Is x < a[4] No

Is x > a[4] Yes

low = mid + 1 = 5

$T(n) = 2T(n/2) + n$ Solve using Recursion Tree.



at K^{th} level problem Size = 1

at K^{th} level problem size = $\frac{n}{2^k}$

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

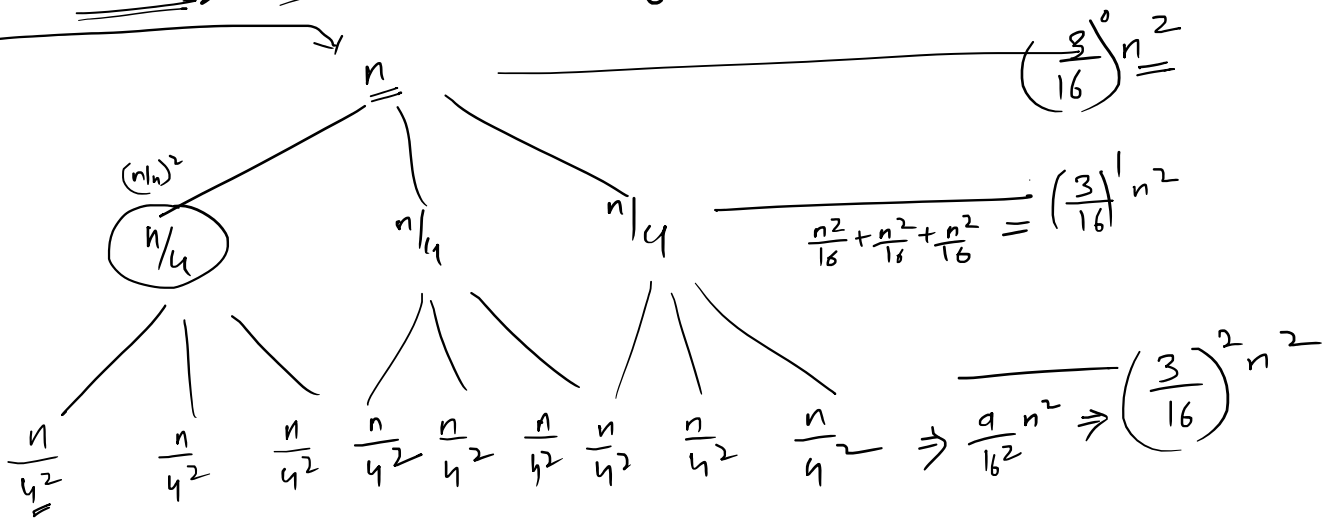
$K = \log_2 n$

Cost of Every level = n

No of levels = $k = \log_2 n$

Total Cost = $n \log_2 n$

$T(n) = 3T(n/4) + n^2$ Solve using Recursion Tree



At k^{th} level problem size = 1

At k^{th} level problem size = $\frac{n}{4^k}$

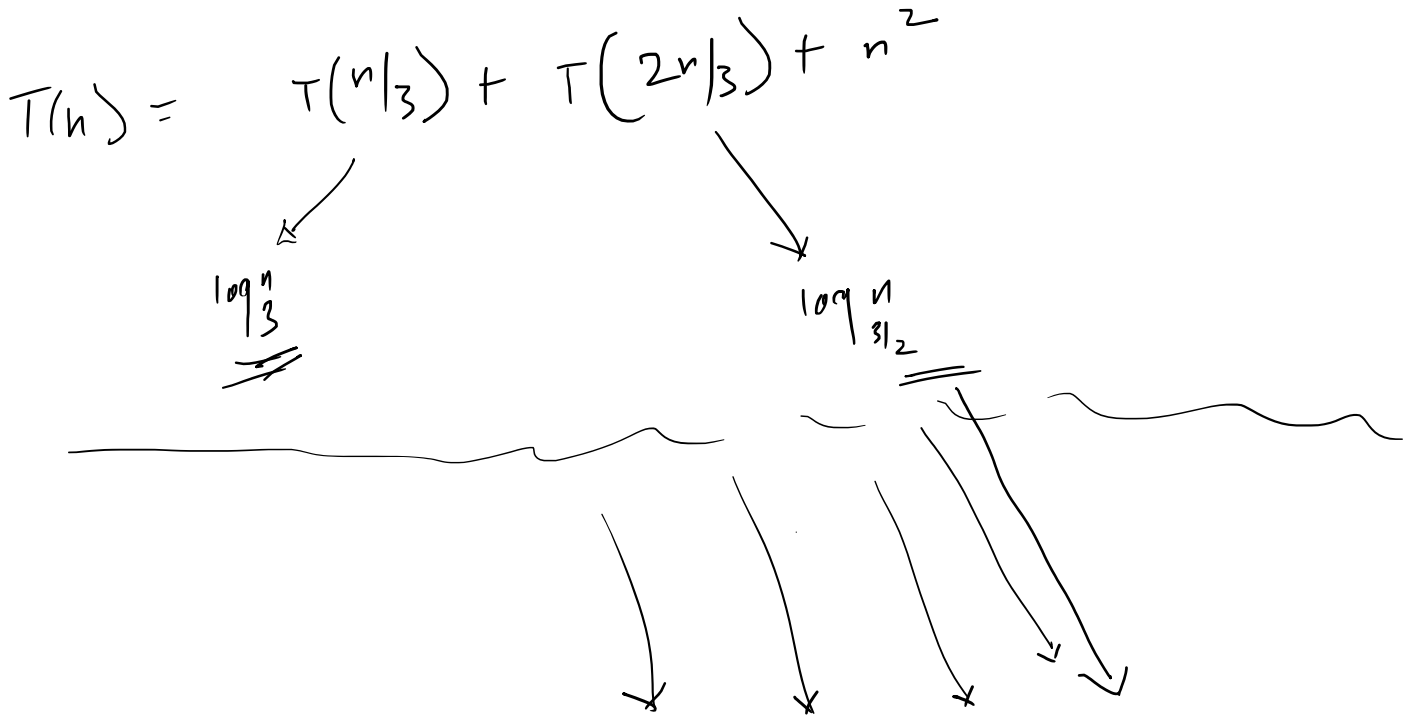
$\frac{n}{4^k} = 1$ $\quad 4^k = n$

$$k = \log_4 n$$

Total cost $= n^2 \left((\frac{3}{16})^0 + (\frac{3}{16})^1 + (\frac{3}{16})^2 + \cdots (\frac{3}{16})^k \right)$

$= n^2 \left((\frac{3}{16})^0 + (\frac{3}{16})^1 + (\frac{3}{16})^2 + \cdots (\frac{3}{16})^{\log_4 n} \right)$

$= n^2 \left[\frac{1 - (3/16)^{\log_4 n}}{1 - (3/16)} \right] \leq n^2$


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

(1) Substitution Method.

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

(2) $T(n) = 1$ if $n = 1$
$= 8T(n/2) + n^2$ if $n > 1$

(3) $T(n) = 1$ if $n = 2$
$= 7T(n/2) + n^2$ if $n > 2$

Recursive Tree

(1) $T(n) = T(n/5) + T(4n/5) + n$.

(2) $T(n) = T(n/10) + T(9n/10) + n$

(3) $T(n) = T(n/5) + T(3n/5) + n$

(4) $T(n) = T(n/2) + T(n/3) + T(n/4) + c$

Master Method

Master Method

(1) $T(n) = 8T(n/2) + n^2$

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

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

(4) $T(n) = 32T(n/2) + n^3$

(5) $T(n) = T(\sqrt{n}) + c$

Substitution → $T(n) = T(n^{1/2}) + c.$

put $n = 2^k$

$T(2^k) = T(2^{k/2}) + c.$

let $T(2^k) = S(k)$

$\Rightarrow \boxed{S(k) = S(k/2) + c}$

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

$a = 1, \quad b = 2, \quad f(k) = c.$

$k^{\log_b^a} = k^{\log_2^1} = n^0 = c$

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

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

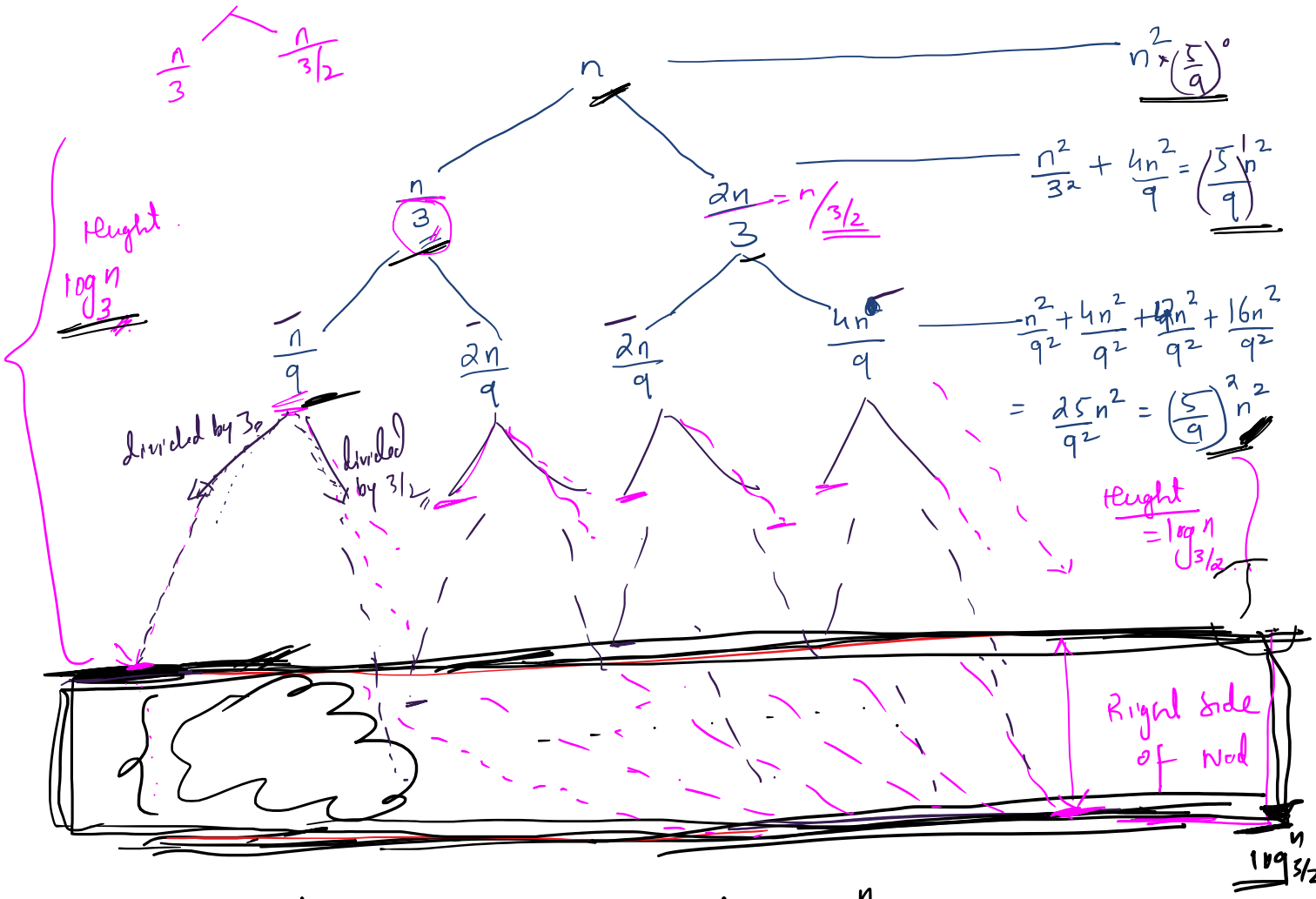
$= O(k^0 \log k)$.

$= O(\log k)$

But $n = 2^k$ $\therefore k = \log_2 n$

$= O(\log \log_2 n)$

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



Consider left side = The height of left side = $\log_3 n$

Cost on left side = $\left(\frac{5}{9}\right)^0 n^2 + \left(\frac{5}{9}\right)^1 n^2 + \left(\frac{5}{9}\right)^2 n^2 + \cdots \left(\frac{5}{9}\right)^{\log_3 n} n^2$

$= n^2 \left[\left(\frac{5}{9}\right)^0 + \left(\frac{5}{9}\right)^1 + \left(\frac{5}{9}\right)^2 + \cdots \left(\frac{5}{9}\right)^{\log_3 n} \right]$

$\frac{1-a^n}{1-r}$

$= n^2 \left[\frac{1 - (5/9)^{\log_3 n + 1}}{1 - 5/9} \right] = \frac{9n^2}{4} \left(1 - (5/9)^{\log_3 n + 1} \right)$

$= (n^2)$

Here the cost is calculated for the height = $\log_3^n$ only.
Right side is not considered.
So this cost < Actual Total cost

$n^2 < T(n)$

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

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

The lower side Height gives Lower Bound (Best Case)

Right Side → The max Height = $\log_{3/2}^n$.

Total cost = $\left(\frac{5}{9}\right)^0 n^2 + \left(\frac{5}{9}\right)^1 n^2 + \left(\frac{5}{9}\right)^2 n^2 + \left(\frac{5}{9}\right)^3 n^2 \cdots + \left(\frac{5}{9}\right)^{\log_{3/2}^n} n^2$.

$= n^2 \left[\frac{1 - (5/9)^{\log_{3/2}^n + 1}}{1 + 5/9} \right] = \frac{9n^2}{4} \left(1 - (5/9)^{1 \times \log_{3/2}^n + 1} \right)$

$= n^2$

Here we have considered the Height till $\log_{3/2}^n$
But there Missing nodes in left side that stopped at $\log_3^n$

Actual Total Cost < Calculated Cost Considering Right Side

$T(n) < n^2$

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

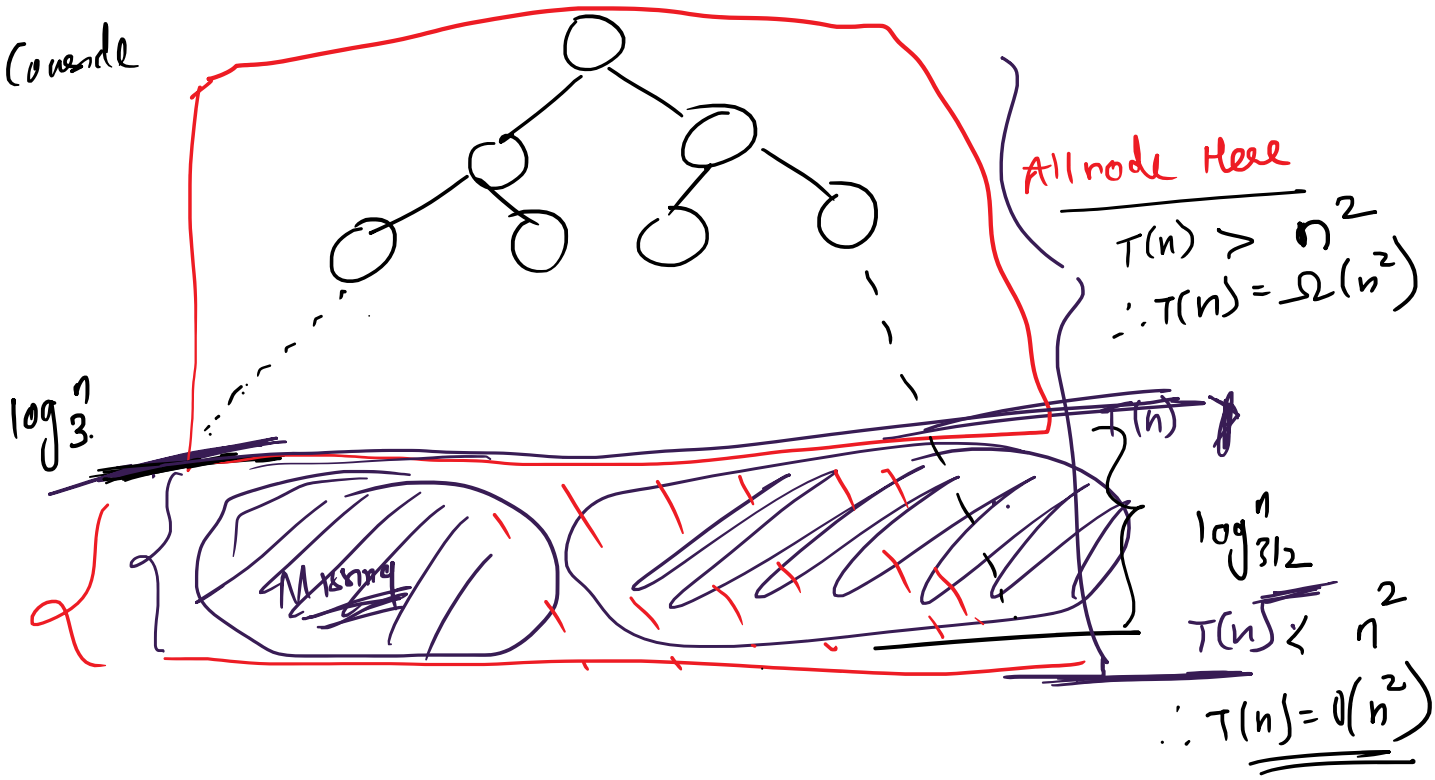
Note
$f(n) = O(g(n))$
$f(n) \leq g(n)$.

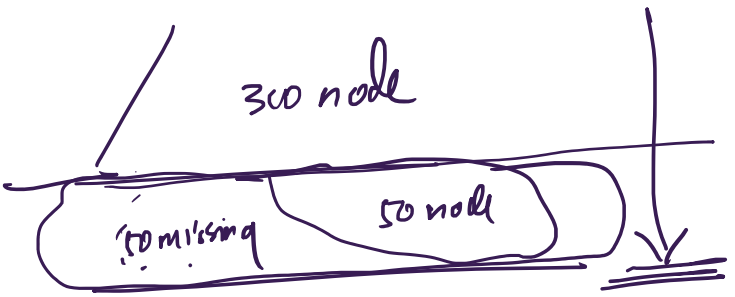
Right Side (Side with More Height) gives Upper Bound

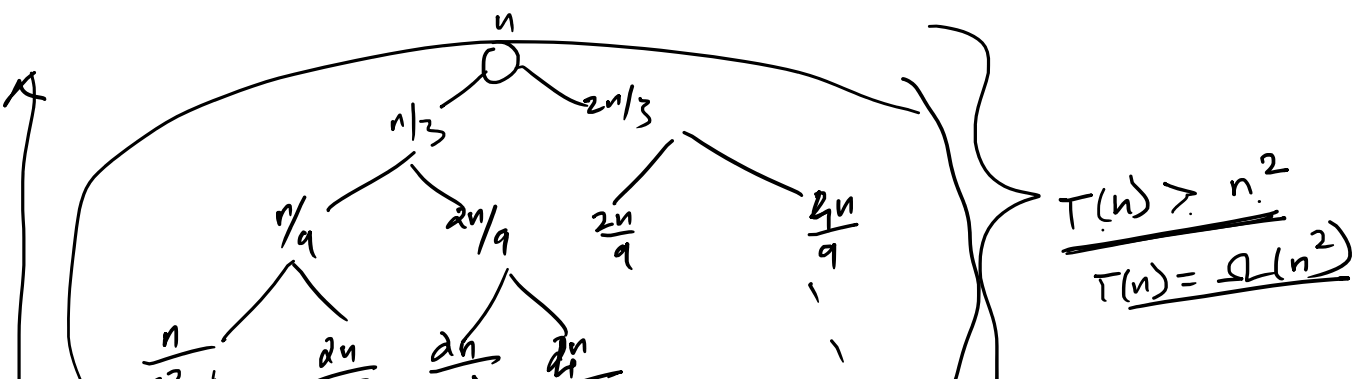
Right Side (Side with more Height) gives Upper Bound

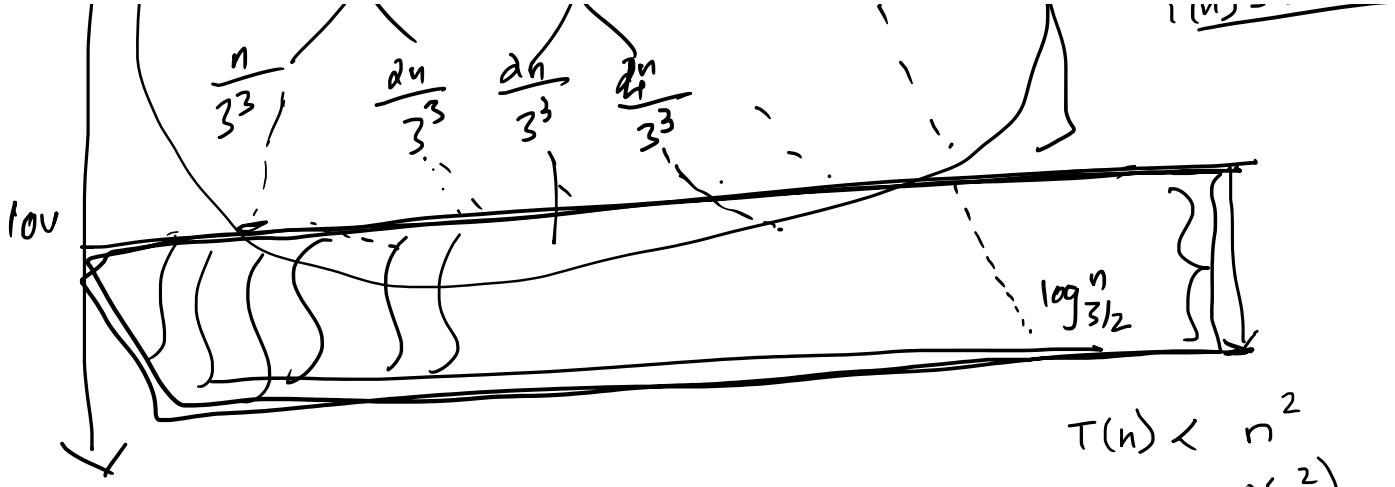
$\therefore \quad T(n) = \Omega g(n^2) \qquad \therefore \quad T(n) = O(n^2)$

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



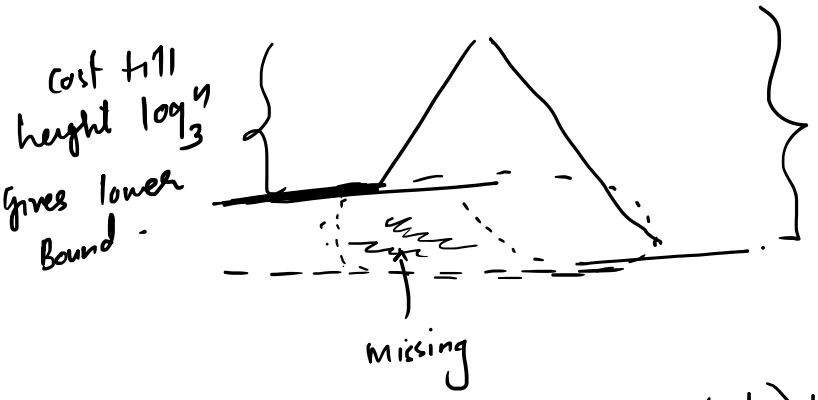






$T(n) < n^2$

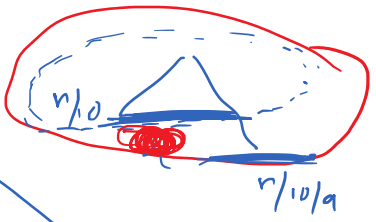
$T(n) = O(n^2)$



cost till height $\log_{3/2} n$ gives upper bound.

Quicksort $T(n) = 2T(n/2) + n$ — partition

Binary Search. $T(n) = T(n/2) + 1$ — mid overhead



$T(n) = T(n/10) + T(9n/10) + n$

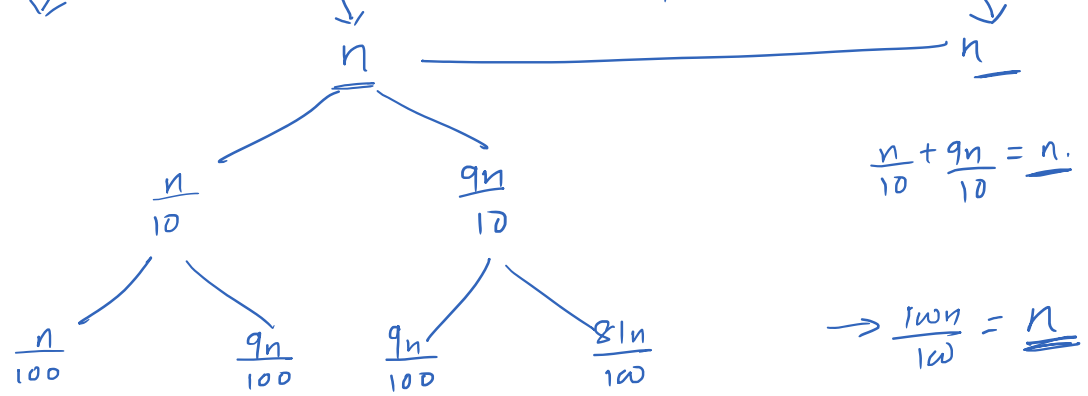
$T(n/10)$ divided by 10

$T(9n/10)$ divided by $10/9$

$1 = \log_{10/9} n$

Height $= \log_{10} n$

$\log_{10/9} n$



$\frac{n}{10} + \frac{9n}{10} = n$

$\rightarrow \frac{100n}{100} = n$

Consider left side $\Rightarrow$ Height $= \log_{10} n$

Cost of each level $= n$

$T(n) > n \log_{10} n$

$T(n) = \Omega(n \log_{10} n)$

$= \Omega(n \log n)$

Consider Right Side = Height = $\log_{10/9} n$

Cost of each level $= n$.

Computed Cost = $n \log_{10/9} n$

$T(n) < n \log_{10/9} n$

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

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

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

③ Master Method → gf is cook book method to determine Time Complexity of Recurrence relation of form

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

Here a and b are +ve constants

$a \geq 1$ & $b > 1$

$f(n)$ = is +ve fn.

Q. T(n) can be bounded asymptotically as follows →

Case 1) gf $f(n) = O(n^{\log_b^a - \epsilon})$ where $\epsilon > 0$

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

Case 1)

$f(n) < n^{\log_b^a}$

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

Case 2) gf $f(n) = \Theta(n^{\log_b^a})$ Then

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

Case 2) gf $n^{\log_b^a} = f(n)$

$T(n) = O(n^{\log_b^a} \log_2 n)$

Case 3) gf $f(n) = \Omega(n^{\log_b^a + \epsilon})$ ϵ is +ve constant

and if $af(n/b) \leq c \cdot f(n)$ for $c < 1$

Then $T(n) = \Theta(f(n))$

Case 3)

gf $n^{\log_b^a} < f(n)$

Then we have to prove for $c < 1$ $af(n/b) \leq c \cdot f(n)$

gf proved then $T(n) = O(f(n))$

Steps. gf $n^{\log_b^a} > f(n)$ ⟶ $O(n^{\log_b^a})$

gf $n^{\log_b^a} = f(n)$ ⟶ $O(n^{\log_b^a} \log n)$.

gf $n^{\log_b^a} < f(n)$ ⟶ $O(f(n))$.

gf proved

$af(n/b) \leq c \cdot f(n)$

for $c < 1$

Trick

① Identify a, b, $f(n)$ from given relation

② Calculate $n^{\log_b^a}$

③ Compare $n^{\log_b^a}$ with $f(n)$

Whoever is greater by polynomial time, it is the Answer.

④ Agar same Hai then $n^{\log_b^a} \log n$

⑤ gf one side is not greater by polynomial time then make both side same.

then apply this

$T(n) = a\,T(n/b) + f(n)$

① $T(n) = 8\,T(n/2) + n^2$

a b $f(n)$

$a = 8, \quad b = 2, \quad f(n) = n^2$

(1) $n^{\log_b^a} \Rightarrow n^{\log_2^8} \Rightarrow n^3$

compare with $f(n) = n^2$

✓ $n^{\log_b^a} > f(n)$

Case 1 satisfied $\longrightarrow T(n) = O(n^{\log_b^a}) = O(n^3)$

② $T(n) = 1 \cdot T(n/2) + c$

↑ constant (any constant value)

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

$n^{\log_b^a} = n^{\log_2^1} \Rightarrow n^0 = 1 = \text{constant}$

$\therefore n^{\log_b^a} = f(n)$ condition 2 satisfied =

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

$= O(c \cdot \log_2 n) = O(\log n)$

Bubble Sort ->

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
void bubblesort(int x[],int n)
{
   int i,j,t;
   for(i=0;i<n-1;i++)
   {
      for(j=0;j<n-1;j++)
      {
         if(x[j]>x[j+1])
         {
            t=x[j];
            x[j]=x[j+1];
            x[j+1]=t;
         }
      }
   }
}


int main()
{
   int *inp,i,j;
   FILE *input,*output;
   float total;
   clock_t start,end;

      printf("Bubble sort\n");

      input = fopen("input.txt","w");
      output = fopen("output.txt","w");

      for(i=5000;i<=25000;i=i+5000)
      {
         inp = (int*)(malloc(i*sizeof(int)));

         printf("\nFor n = %d\n",i);
         //write i numbers of randam values from 0 to i in input file
               for(j=0;j<i;j++)
         {
            fprintf(input,"%d ",rand()%i);
         }
         fclose(input);

         input = fopen("input.txt","r");

//read the i numbers of random values and store them in array of int
               for(j=0;j<i;j++)
         {
            fscanf(input,"%d",&inp[j]);
         }
```

File Handling

* To access file we need FILE pointer

* To open file : FILE * input, * output

input = fopen("Filename", "mode")

will open file in given mode and will return the add of file

mode → r = will open file read mode
w = " " " write mode (override)
a = append mode

To read form file →

fscanf ("pointername", "%d", "array add")

pointername → kahan se
%d → kya type read karna hai
array add → read karke kahan store karna hai

To write into File

printf("%d", a[i])

fprintf("pointer name", "%d", "kahan k" value)

pointer name → write into this file
%d → kya type
kahan k value

int *inp → pointer to integer
FILE *input,*output; → File pointers
clock_t start,end; → variables of type clock_t
input = fopen("input.txt","w"); → write mode
output = fopen("output.txt","w"); → write mode
for(i=5000;i<=25000;i=i+5000) → i = 5000, 10000, 15000, 20000, 25000 → 5 time
inp = (int*)(malloc(i*sizeof(int))); → inp
fprintf(input,"%d ",rand()%i); → Writes into file → Write random values in input file
fclose(input); → Closed input.txt
input = fopen("input.txt","r"); → open in read mode.
fscanf(input,"%d",&inp[j]); → read int value → place the value in array. → read from file

```
        fscanf(input,"%d",&inp[j]);
    }
    start=clock();
    bubblesort(inp,i);
    end=clock();
    total=(float)(end-start)/CLOCKS_PER_SEC;
    for(j=0;j<i;j++)
    {
    fprintf(output,"%d ",inp[j]);
    }
    printf("Total Time : %f\n\n",total);
}
 fclose(input);
fclose(output);
return 0;

}
```

read from file

start counting time.

call Bubble Sort fn

stop Counting time

No of clock

Total seconds

total clocks / constant

Write content of inp array in the output.txt file.

Write

(float)(end - start) ⇒ No of clocks spent bet^n end & start (CPU clock cycle)

No of sec = No of clocks spent / CLOCKS_PER_SEC.

constant (predefined in <time.h>)

* will vary from system to system.

Selection Sort ->

```
void selectionsort(int a[],int n)
{
    int i,j,t,min,p;
    for(i=0;i<n-1;i++)
    {
        min=a[i];
        p=i;

        for(j=i+1;j<n;j++)
        {
            if(a[j]<min)
            {
            min=a[j];
            p=j;
            }

        }
        t=a[p];
        a[p]=a[i];
        a[i]=t;
    }
}
```

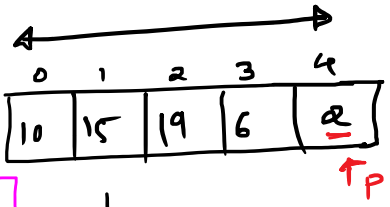
let min → smallest element
p ⇒ posn of smallest element

will give min value and its posn from i+1 to n

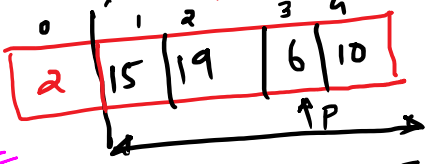
Swapping a[p] a[i]

i = 0
min = a[0] = 10
p = 0

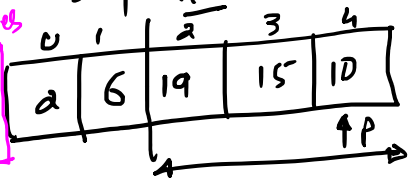
j = 1 | j = 2 | j = 3 | j = 4
a[1] < min | a[2] < min | a[3] < min | a[4] < min
NO | | Yes | 2 < 6 Yes
| | min = a[3] = 6 | min = 2
| | p = 3 | p = 4



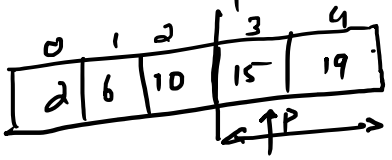
1st Iteration
Swap a[0] & a[p]



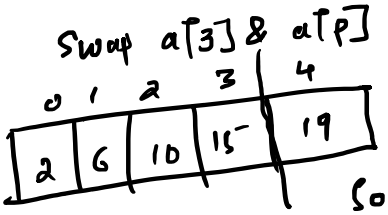
Swap a[1] & a[p]



Swap a[2] & a[p]



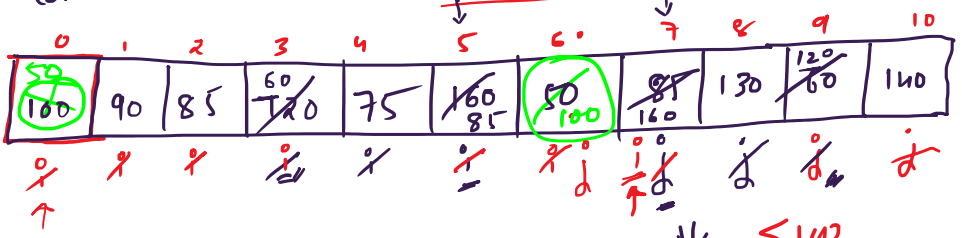
Swap a[3] & a[p]



Sorted

QuickSort :-> Consider. logic of Partition.

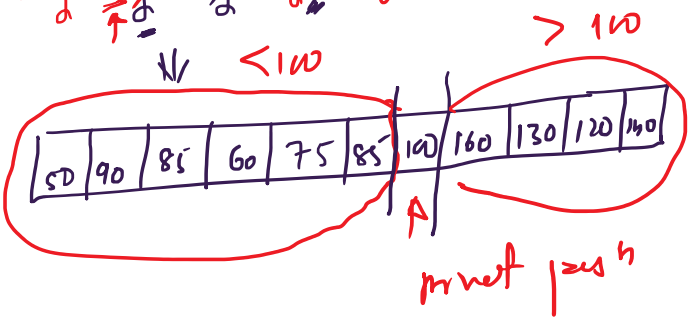
n=11



x = a[low] = 100

i = 0 (low) → i will search for element > x

j = 10 (high) → j will search for element < x




```
while (i<=j)
{ while (a[i]<=x)
    i++;
}
```
i will stop when a[i] > x.

```
while (a[j] > x)
    j--;
}
```
j will stop when a[j] < x

```
if (i < j)
{ t = a[i];
  a[i] = a[j];
}
a[j] = t;
}
```
Swap a[i] & a[j]

```
// swap a[low] & a[j]
t = a[low];
a[low] = a[j];
a[j] = t;
```

Return j //

from the pos^n the array is divided into 2 parts.

Repeat logic on each part recursively.

Bubble Sort → Brute Force Approach

n = 64

① 63 comp

② 62 comp

.

.

.

63 iteration. 1 comp

No of Com : 1 + 2 + 3 + . . . 64

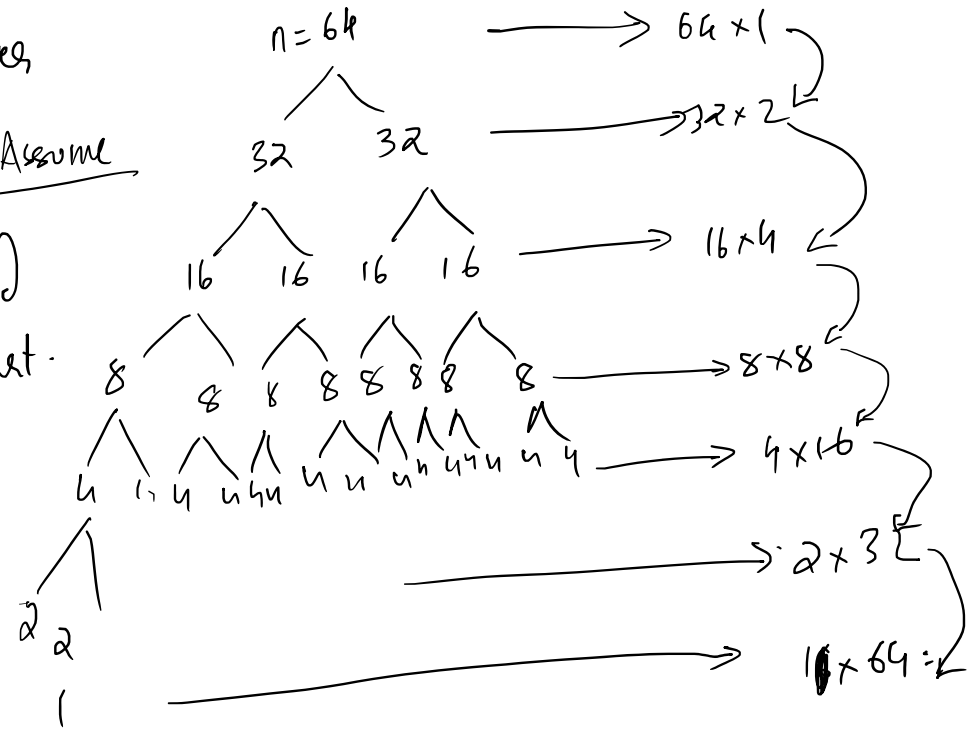
$\frac{64 * 65}{2} = 32 * 65$

Divide & Conquer

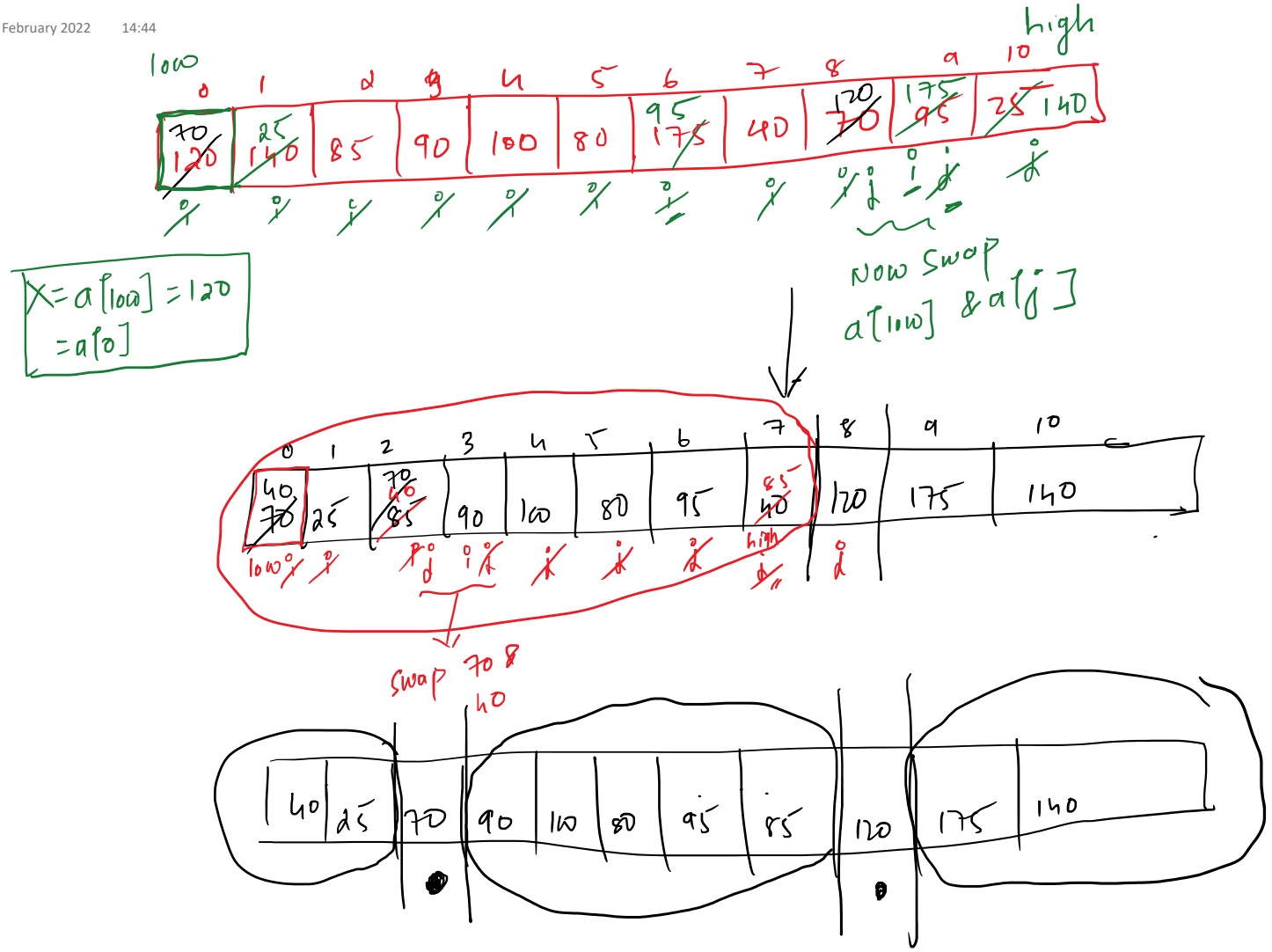
Assume

Best Case

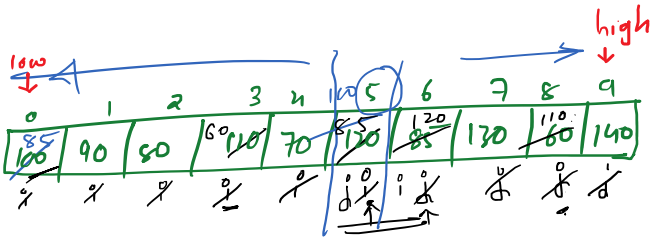
Array divided into Equal part.



If $n = 2^x$ x iterations.


high
100
0 1 2 3 4 5 6 7 8 9 10
70 25 85 90 100 80 95 40 120 175 140
X=a[100] =120
=a[0]
Now swap
a[100] & a[j]
0 1 2 3 4 5 6 7 8 9 10
40 25 70 90 100 80 95 85 120 175 140
swap 70 & 40
40 25 70 90 100 80 95 85 120 175 140

Quicksort →




```
#include<stdio.h>
int partition(int a[],int low,int high)
{
int x,i,j,t;
x=a[low];
i=low;
j=high;

   while(i<=j) // jab tak i & j cross nahi ho jate
   {
      while(a[i]<=x)//i is looking for greater value than x
      i++;

      while(a[j]>x)//j is looking for <= value than x
      j--;
      // after i & j stop
      if(i<j)
      {

      t=a[i];
      a[i]=a[j];
      a[j]=t;

      }
   }

   t=a[low];
   a[low]=a[j];
   a[j]=t;

   return j;
   }
```

for(i=low, j=high; i<=j;)

x = a[low]

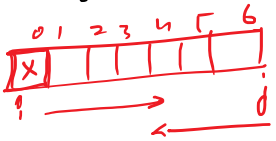
x = 100

i = 0

j = 9

i<j yes

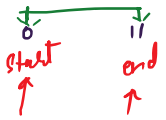
i<j yes



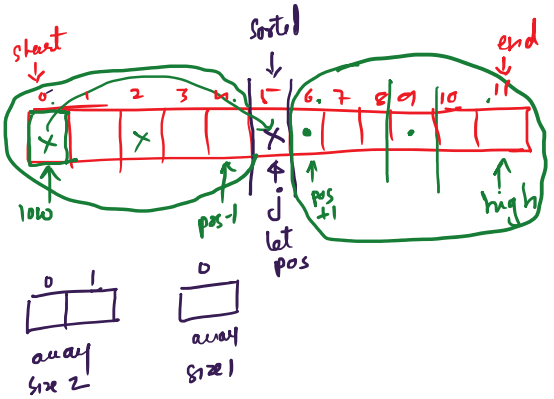
n comparison to i to become > j

swap a[i] & a[j]

swap a[low] & a[j]

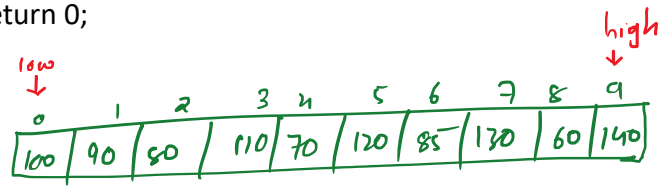



```
void quicksort(int a[],int low,int high)
{
int pos;
if(low<high)//array ka size 1 nahi hojata
{
   pos=partition(a,low,high);
   quicksort(a,low,pos-1); → 0 to pos-1
   quicksort(a,pos+1,high); → pos+1 → high

}
}


int main()
{
```



```
    int a[20],n,i;
    printf(" enter nos of elements in array\n");
    scanf("%d",&n);

    printf("enter elements in array\n");
    for(i=0;i<n;i++)
    scanf("%d",&a[i]);

    printf(" original order of elements\n");
    for(i=0;i<n;i++)
    printf("%d ",a[i]);

    printf("\n");

    quicksort(a,0,n-1);

    printf(" sorted order of elements\n");
    for(i=0;i<n;i++)
    printf("%d ",a[i]);

    return 0;
}
```

array · start index · end index



X = a[low]

X = 100

Vimp Gate

Extended Master Theorem →

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

Here $a \geq 1$, $b > 1$, $k \geq 0$, p = real no

Case ① if $a > b^k$ ⟶ $T(n) = \Theta(n^{\log_b a})$.

Case ② if $a = b^k$

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

② if $p = -1$ ⟶ $T(n) = \Theta(n^{\log_b a} \log \cdot \log n)$.

③ if $p < -1$ ⟶ $T(n) = \Theta(n^{\log_b a})$.

Case 3

③ if $a < b^k$

① if $p \geq 0$ ⟶ $T(n) = \Theta(n^k (\log n)^p)$

② if $p < 0$ ⟶ $T(n) = O(n^k)$.

Ex $T(n) = T(\sqrt{n}) + c$ ⟶ To convert into master

=

Practise:

① $T(n) = aT(n/2) + c$

⑤ $T(n) = 8T(n/2) + n^2 \log n$

⑥ $T(n) = 7T(n/2) + n^2$

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

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

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

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

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

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

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

(9) $T(n) = aT(n/2) + \frac{n}{\log n}$.

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

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

$a=2$, $b=2$, $K=1$, $P=-1$

$a=2$

$b^K \Rightarrow 2^1 = 2$

$a = b^K$. (case 2)

Chk P $P=-1$

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

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

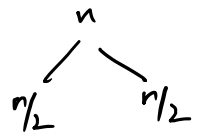
$= \Theta(n \log(\log n))$

Divide & Conquer →

Approach →

① Divide the original problem into subproblems.

Ex



② Conquer Recursively
i'e Recursively solve Each subproblem.

③ Combine the solution of subproblem to get the solution of main/original problem.

```
DAC(a, i, j)
{
  if (small(i,j))          } if problem size is very small then return directly
  return solution small(i,j)
  else
  { mid = (i+j)/2;         } Divide  O(1)   Here assumed center as partition point.
  subproblems {
    b = DAC(a, i, mid);    }
    c = DAC(a, mid+1, j);  } Recursively solving subproblems & getting their solution
  d = combine(b,c);        } Combining soln of subproblem
  return d;
}
```

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

5

* Best Case Recurrence Rel^n for QuickSort → Assume the partition f^n divides array into 2 halves.

$$T(n) = \underbrace{2\,T(n/2)}_{\text{Recursion call}} + n \quad \leftarrow \text{partition } f^n \text{ cost.}$$

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

$a = 2$.
$b^k \Rightarrow 2^1 = 2$ $\qquad a = b^k$ case 2

Chk $p = 0 > -1 \implies T(n) = \Theta(n^{\log_b a} (\log n)^{p+1})$

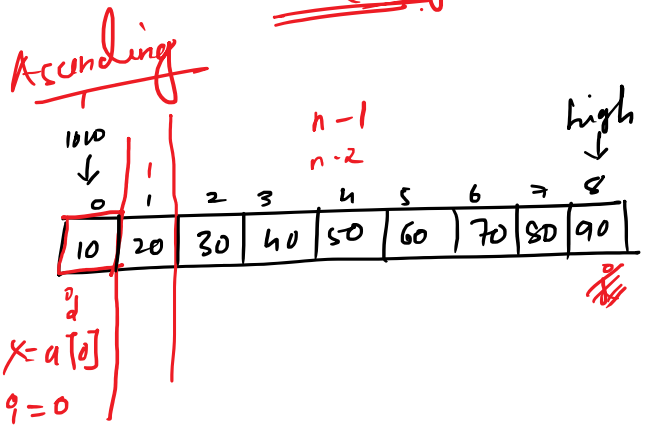
$= \Theta(n^{\log_2 2} (\log n)^{0+1})$

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

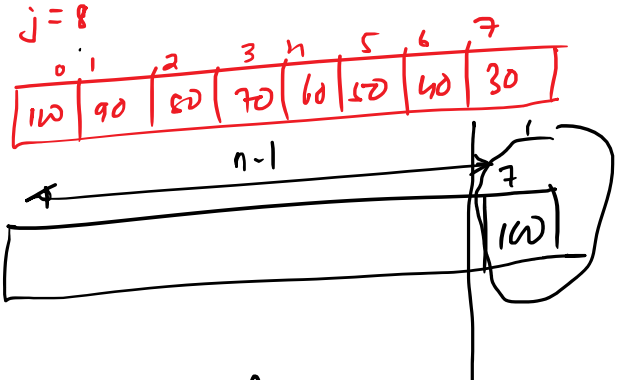
Worst Case for QuickSort →.

If input array is Sorted (AS / DS)

Ascending



Descending



Here After 1st Iteration array is divided into size 1 & n-1

— 2nd — 1 & n-2

∴ Recurrence Rel^n = $T(n) = T(n-1) + n$. ← Time for partition

∴ Recurrance Rel^n = $T(n) = T(n-1) + n$ ← Time for partition

for Input Sorted

Substitution

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

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

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

1

$n-k=1$

$n = 1+k$

$k = n-1$

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

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

$= 1 + 2 + 3 + \cdots n$

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

∴ To QuickSort

If Input is Sorted Order $\xrightarrow{\text{worst case}}$ $O(n^2) \Rightarrow O(n^2)$

If Input is Randomly array $\xrightarrow{\text{Best case}}$ $O(n \log n) = \Omega(n \log n)$

Recursive Binary Search. ->

```
#include<stdio.h>
int binarysearch(int a[],int low,int high,int x)
{
    int mid;
    if(low<=high)
    {
        mid=(low+high)/2;

        if(x==a[mid])
        return 1;
        if(x<a[mid])
        return binarysearch(a,low,mid-1,x);
        else
        return binarysearch(a,mid+1,high,x);
    }
    return 0;
}

int main()
{
    int a[20],n,i,x;
    printf(" enter nos of elements in array\n");
    scanf("%d",&n);

    printf("enter elements in array\n");
    for(i=0;i<n;i++)
    scanf("%d",&a[i]);

    printf(" enter element to search\n");
    scanf("%d",&x);


    printf("elements in array\n");
    for(i=0;i<n;i++)
    printf("%d ",a[i]);

    printf("\n");
```

O(1).

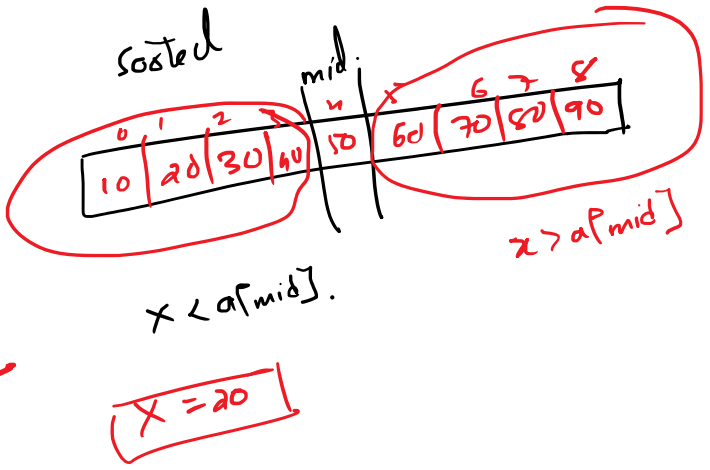
T(n/2)

In Non Recursive high = mid-1

high

In Non Recursive low = mid+1

low



start last to search.

```
    if(binarysearch(a,0,n-1,x))
    printf(" %d element is present in array\n",x);
    else
    printf(" %d element is not present in array\n",x);


    return 0;
}
```

T.C of Binary Search (Recursive)

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

↖ constant as Time req to calculate mid is constant

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

$a = 1$

$b^k = 2^k = 2^0 = 1$

$a = b^k$ Case 2

chk $p = 0 > -1$

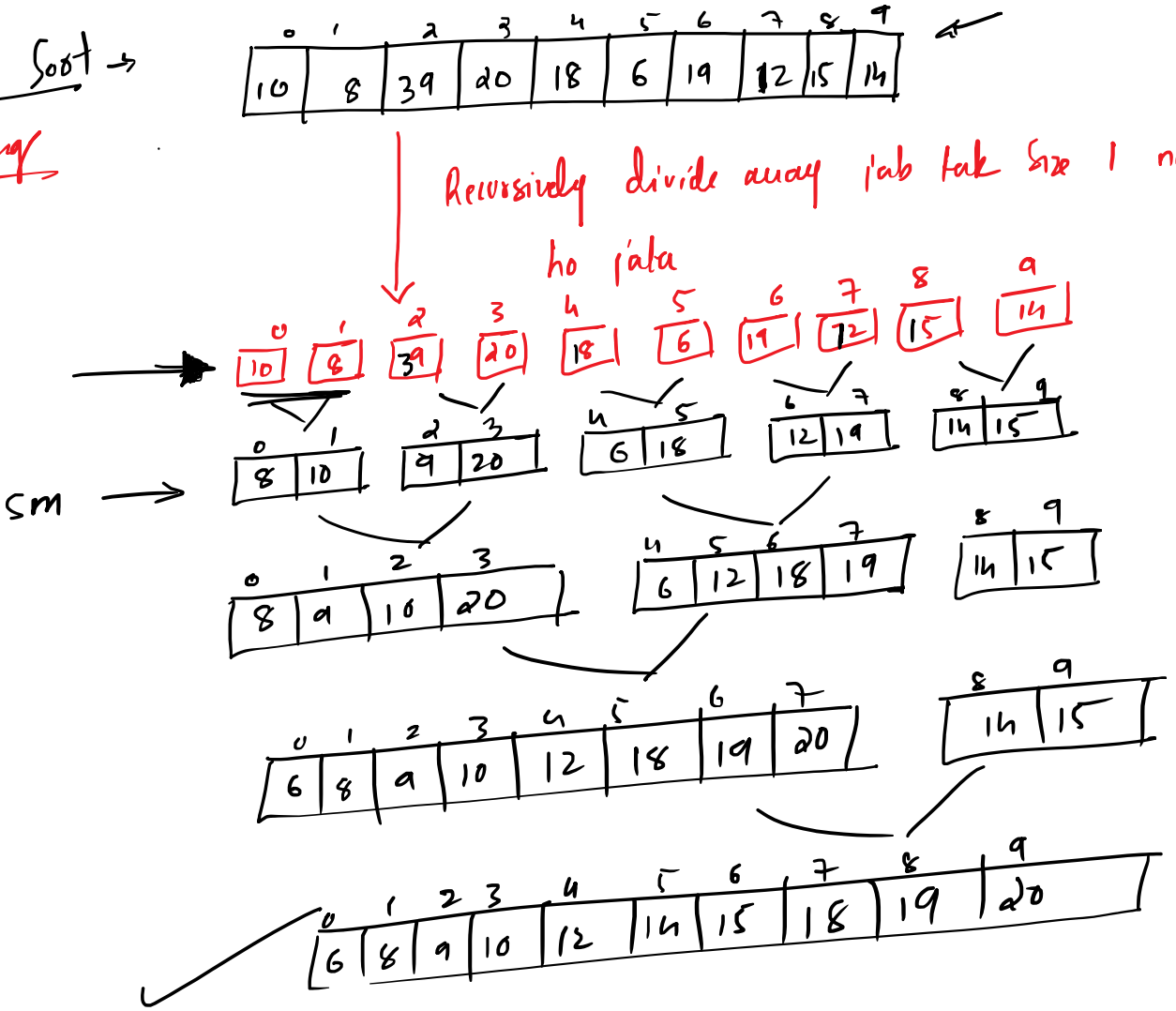
$T(n) = O(n^{\log_b^a} (\log n)^{p+1})$

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

$= O(\log n)$.

Merge Sort →

Working Principle



MergeSort →

```
#include<stdio.h>
void mergesort(int a[],int low,int high)
{
int mid;
if(low<high)
{
 mid=(low+high)/2;
 mergesort(a,low,mid);
 mergesort(a,mid+1,high);
 simplemerge(a,low,mid,high);
}
}

void simplemerge(int a[],int low,int mid,int high)
{
    int temp[50],i,j,k;

    i=low;
    j=mid+1;//start of second portion
    k=0;
    while(i<=mid && j<=high)
    {
        if(a[i]<a[j])
        {
            temp[k++]=a[i++];
        }
        else
        temp[k++]=a[j++];
    }
    //Now chk which part is finished.
    while(i<=mid)//jab tak first portion khatam nahi hota
    temp[k++]=a[i++];

    while(j<=high)//jab tak second portion khatam nahi hota
    temp[k++]=a[j++];

    //copy back from temp to array a
    k=0;
    for(i=low;i<=high;i++)
    a[i]=temp[k++];
    }
```

start. End. main → M

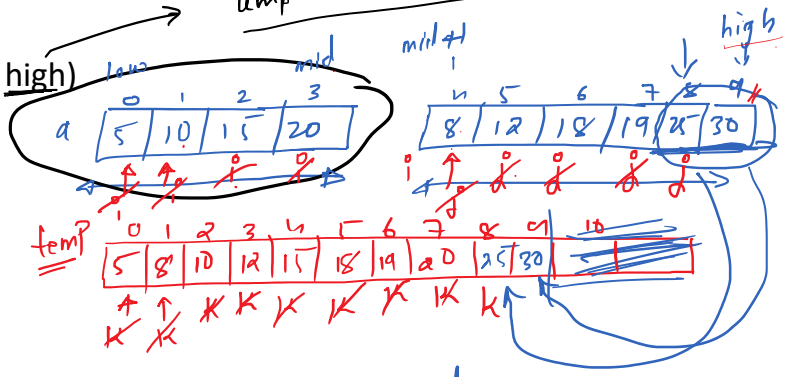
if array size is not one

Calculate Mid.

Recursive Call

Combine sol of Sub problem.

int *temp; temp = (int*) malloc (sizeof(int) * (high+1));



first part End Nahi ho jaata. Second part End Nahi ho jaata.

if false then atleast one part is terminated.

N 0

→ O(n)

Will Execute if Second part is finished.

→ O(n)

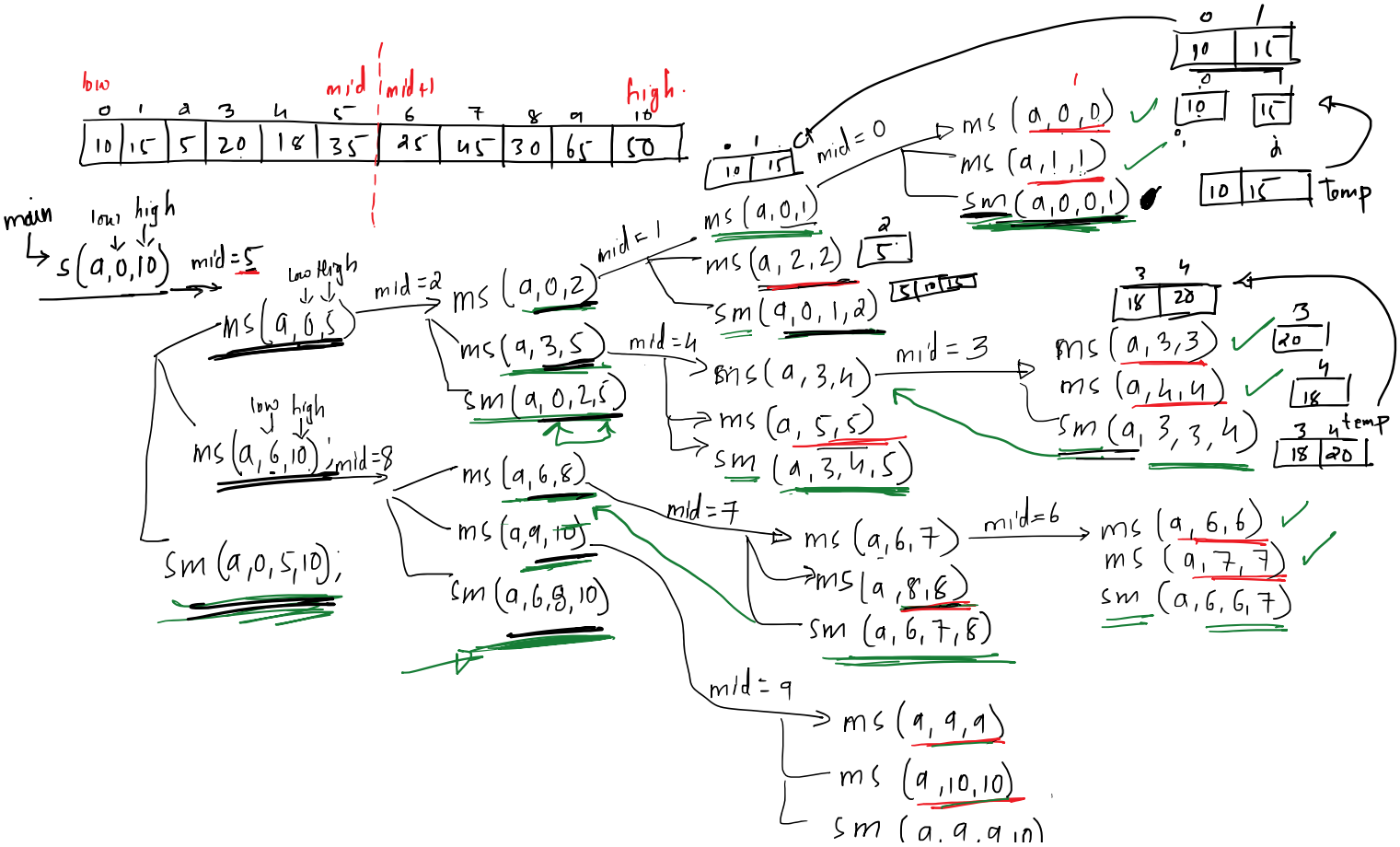
Will Execute if first part is finished → O(n)

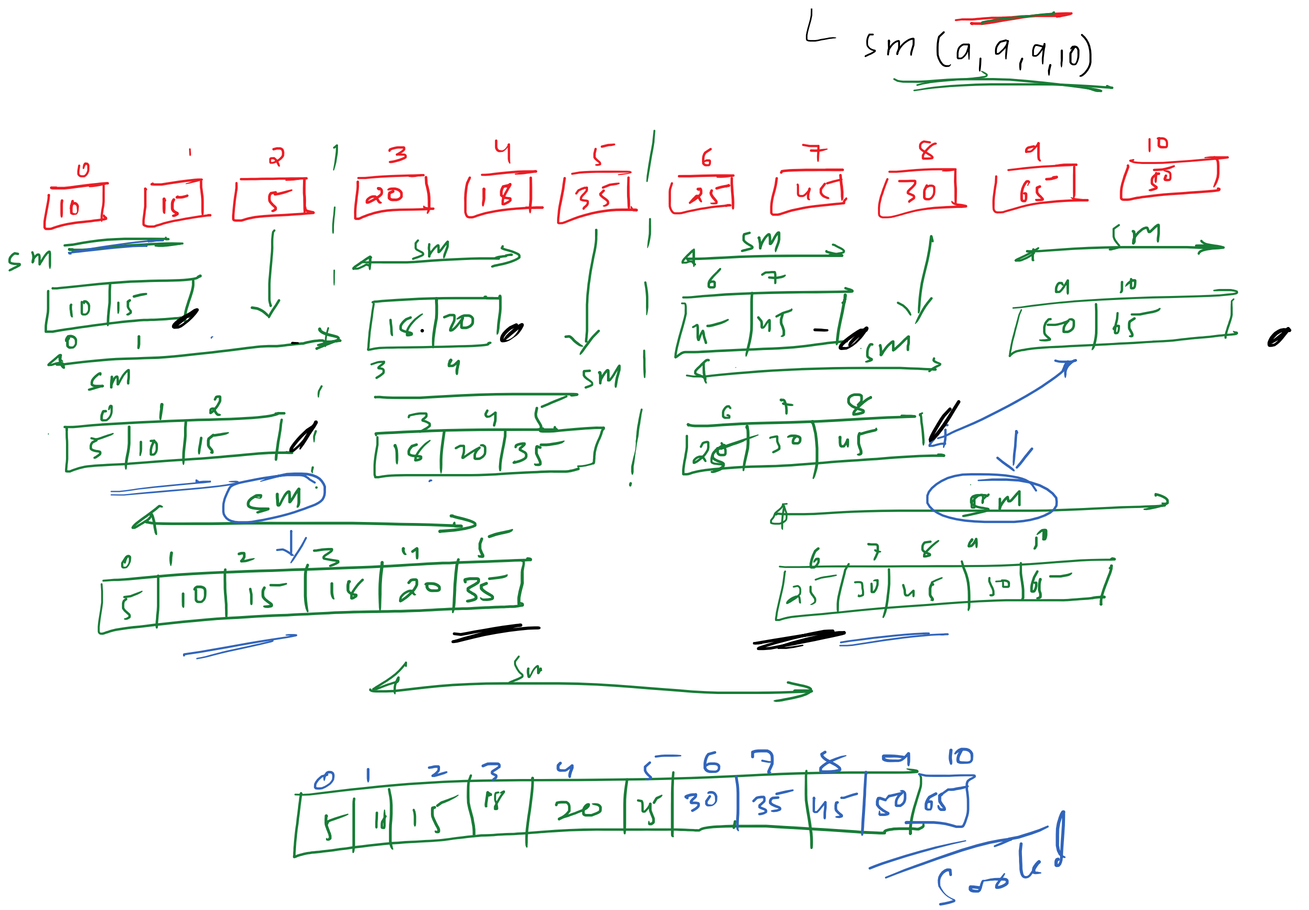
0 11

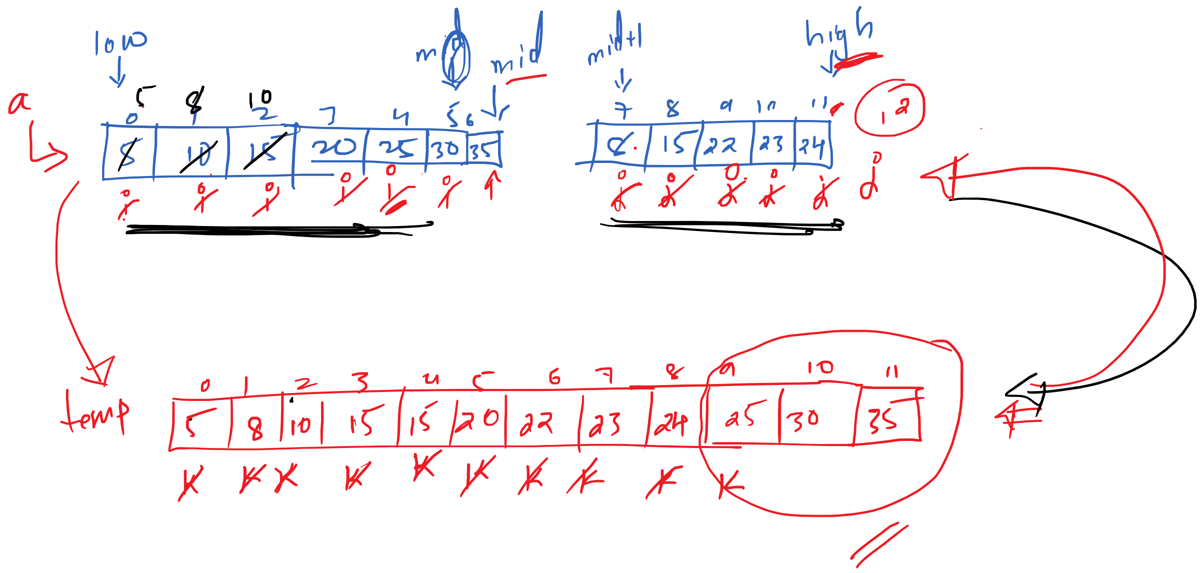
→ O(n)

⇒ n+n+n+n = 4n = O(n)

```
int main()
{
    int a[20],n,i;
    printf(" enter nos of elements in array\n");
    scanf("%d",&n);

    printf("enter elements in array\n");
    for(i=0;i<n;i++)
    scanf("%d",&a[i]);

    printf(" original order of elements\n");
    for(i=0;i<n;i++)
    printf("%d ",a[i]);

    printf("\n");

    mergesort(a,0,n-1);

    printf(" sorted order of elements\n");
    for(i=0;i<n;i++)
    printf("%d ",a[i]);

    return 0;
}
```







low = 0, high = 11, k = 0

for (i = low; i <= high; i++)

a[i] = temp[k++];

Analysis of Mergesort →

Recurrance Relation for Mergesort can be written as.

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

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

```
Mergesort(int a[], int low, int high)
{ int mid;
  if(low < high)
  {
    mid = (low+high)/2 ; → O(1)
    { mergesort(a, low, mid); → T(n/2);
      mergesort(a, mid+1, high); → T(n/2);
      SimpleMerge(a, low, mid, high); → O(n)
```

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

$a = 2$

$b^k = 2^1 = 2$ $\quad a = b^k$

cond^n so chk p.

$p = 0 > -1$ $\quad T(n) = \Theta(n^{\log_b a}(\log n)^{p+1})$

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

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

For Mergesort

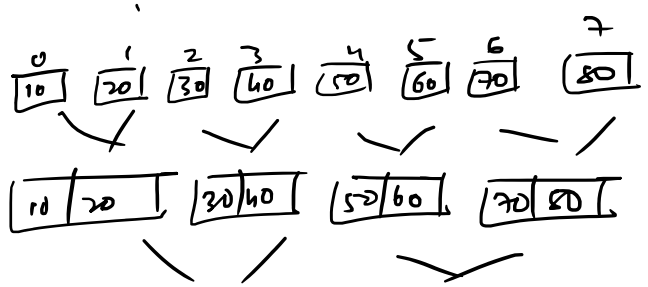
Best Case — $\Omega(n \log n)$

Worst Case — $O(n \log n)$

Avg Case — $\Theta(n \log n)$.

↳

0	1	2	3	4	5	6	7
10	20	30	40	50	60	70	80



finding Min Max Element in an array using Divide & Conquer approach →.

Normal Approach →

```
min = a[0];
for(i = 1; i < n; i++)
    if(a[i] < min)
        min = a[i];
```

⇒ n-1 time

```
max = a[0];
for(i = 1; i < n; i++)
    if(a[i] > max)
        max = a[i];
```

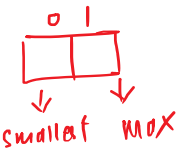
⇒ n-1 time.

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

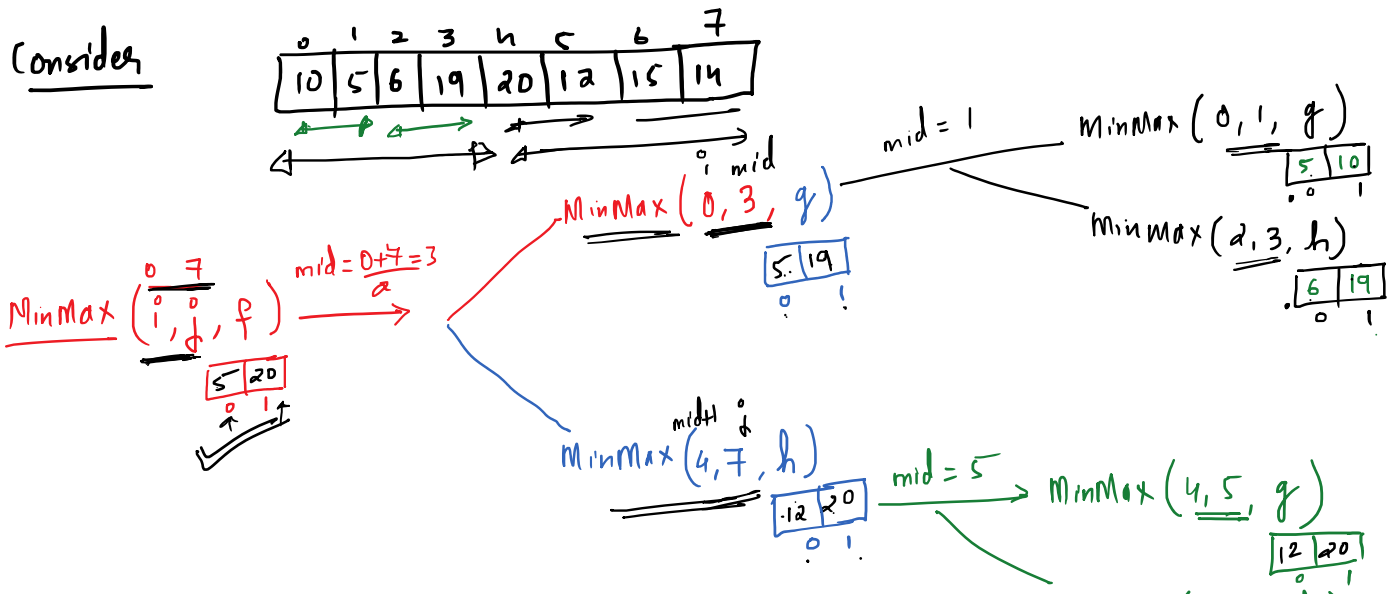
$= 2n - 2$

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

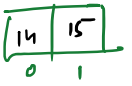
Divide And Conquer Approach →: f is an array of size 2



Consider



MinMax(6, 7, h)



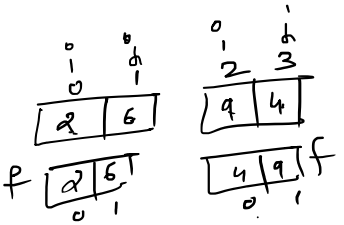
Logic

if (i == j) ⇒ array size is 1

{

f[0] = f[1] = a[i];

}



else if (i == j-1) ⇒ array size is 2

{

f[0] = min(a[i], a[j]);

f[1] = max(a[i], a[j]);

}

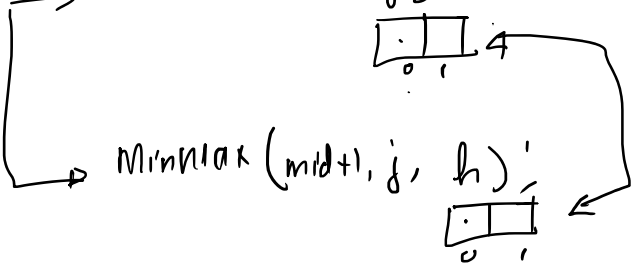
else

{ // array size > 2

mid = (i+j)/2;

MinMax(i, mid, g);

MinMax(mid+1, j, h);



⇒ array of size 2.

f[0] = min(g[0], h[0]);

f[1] = max(g[1], h[1]);

} //.

Module 3 > Greedy Algorithm →

* Greedy Algorithm solves problem which has n inputs and requires us to obtain subsets of solution that satisfies the constraints, is called feasible solution.

* We need to find feasible solution that either maximizes or minimizes the objective fn

* The greedy algorithm works in stages such that only one input is considered at a time.

* The decision will be made at each stage whether the particular input is optimal or not

* If input results into infeasible solution cycle then the input is dropped.

General Algorithm:

```
Greedy (a, n)
{   // a[1...n] possible inputs
    solution = ∅ [Initially empty solution set]

    for (i = 1 to n)
    {
        x = select (a[i])
```

```
{
    x = select(a[i])
    if (feasible(Solution(x))) then
        Solution = Union(Solution, x);
}
return Solution
}
```

Note:

1. we will be given set of n inputs and initially Empty Solution set
2. Consider one input at a time.
   Chk if it makes solution feasible.

   If Yes: Add input to the solution set.
   If NO: Consider the next input and Repeat step 2.

3. Return Solution.

Problems Solved using Greedy Approach.

(1) Minimum Cost Spanning Tree -

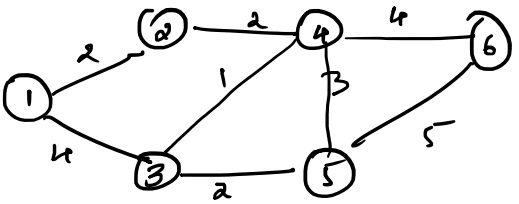
Spanning Tree → It is subgraph of given graph that contains

(i) All the vertices of orginal graph and connected.

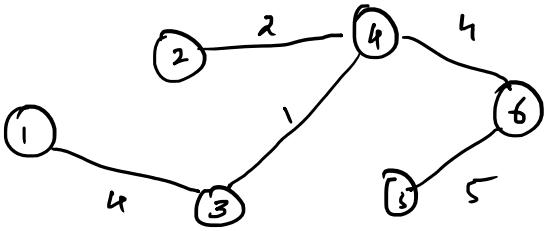
(ii) may not contains all the edges.

(iii) Must not have cycle.

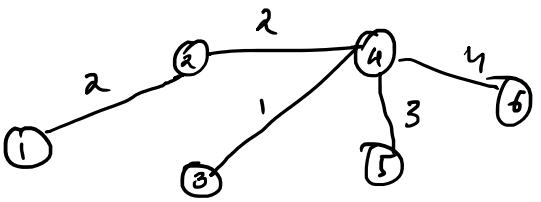
Consider



Possible Spanning Tree'.



Cost = 16



Cost = 12

Many Possible Spanning Tree for given graph →

We want to find Spanning Tree of Minimum Cost.

Known as MCST (Minimum Cost Spanning Tree).

* It is used Network Design

* To find Minimum Cost Spanning Tree
  - Kruskals Algo to find MCST.
  - Prim's Algo to find MCST.

* A graph can have more than one MCST.

Kruskals Algo to find MCST →

* The Algorithm uses set of edges sorted in increasing order of weight/cost
* The Algo starts with Empty Spanning Tree
* Each edge is taken from set of Increasing order of weight and added in the tree.
* An edge is discarded if it creates cycle.

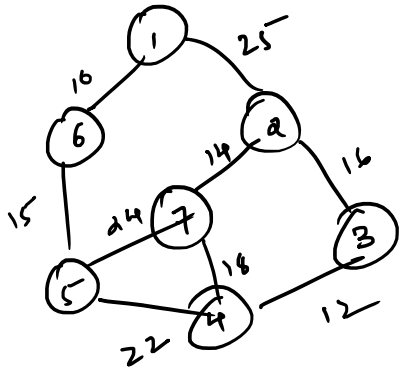
Algo → Given a weighted undirected graph of 'v' vertices and 'e' edges let X be the set of edges sorted in Increasing order of weight.

Let 'g' be graph and 'T' represent Spanning Tree (initially empty).

```
Kruskals(g, T)
{
  while (no of edges in T != V-1).
  {
    let w = next edge from set X
    add(w) in Tree T.
```

```
        if (w forms cycle in T)
            remove w from T.
    }
}
```

Consider graph.

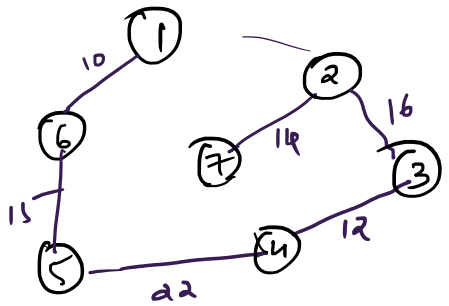


Find MCST using Kruskals Algo →

Step1) Arrange the edge in increasing order of cost/weight

$$X = \left\{ \frac{1-6}{10}, \frac{4-3}{12}, \frac{2-7}{14}, \frac{5-6}{15}, \frac{2-3}{16}, \frac{4-7}{18}, \frac{5-4}{22}, \frac{5-7}{24}, \frac{1-2}{25} \right\}$$

Step2



(1) Consider edge $\frac{1-6}{10}$

It is not making cycle
So it is feasible
add it to Soln set

(2) Consider $\frac{4-3}{12}$

Not making cycle
So feasible
∴ add to Soln set

(3) Consider edge $\frac{2-7}{14}$

Not making cycle
so feasible soln
add to soln set

(4) Consider edge $\frac{5-6}{15}$

Not making cycle
So feasible soln
add to Soln set

(5) Consider edge $\frac{2-3}{16}$

Not making cycle
So feasible soln
add to soln set

(6) Consider edge $\frac{4-7}{18}$

It forms cycle
reject it

(7) Consider edge $\frac{5-4}{22}$

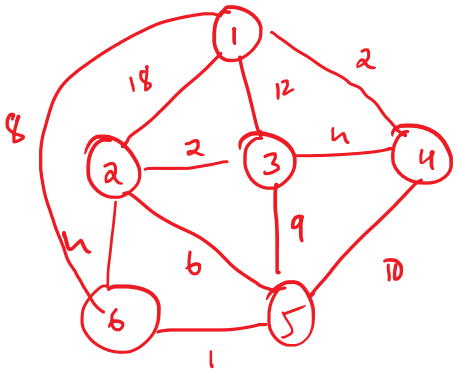
It not forming cycle
feasible soln
add to soln set.

(8) Since No of edges = V-1 (6)

∴ we will stop here as all the edge further will form cycle
So reject it.

Q2) Find MCST Using Kruskals Algo

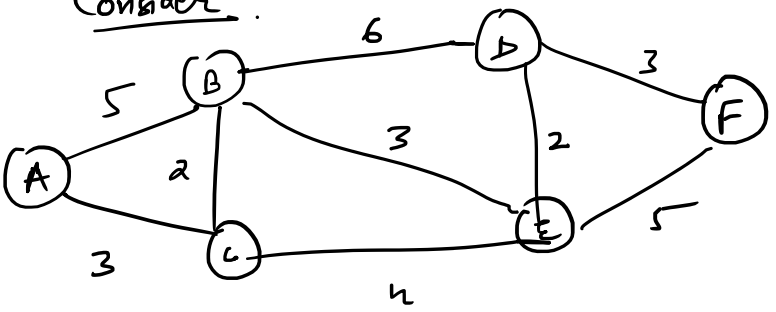
HW



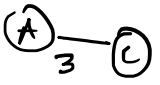
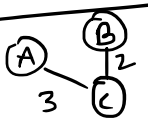
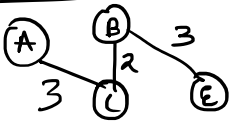
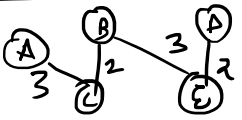
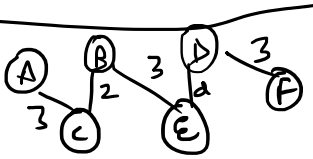
Prims Algorithm to find MCST → Here the Algo is based on Vertex

Method 1

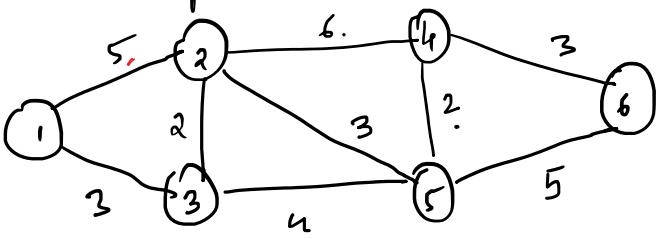
Consider.



Using Prims Algo we find MCST as follows! →

Vertex Selected	Edge Considered	Edge Selected	Cost	Spanning Tree.
A	$\frac{AB}{5}, \frac{AC}{3}$	AC	3	
A, C	$\frac{AB}{5}, \frac{CB}{2}, \frac{CE}{4}$	CB	2	
A, B, C.	$\frac{BD}{6}, \frac{BE}{3}, \frac{CE}{4}$	BE	3	
A, B, C, E	$\frac{BD}{6}, \frac{ED}{2}, \frac{EF}{5}$	ED	2	
ABCDE	$\frac{DF}{3}, \frac{EF}{5}$	DF	3	 MCST.
ABCDEF			Total = 13	

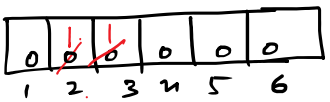
Method 2. → Using Prims to find MST.



current = 1 (vertex) visited[1] = 1, d[current] = 0, count = 1

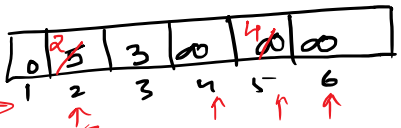
d

P

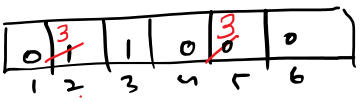


visited

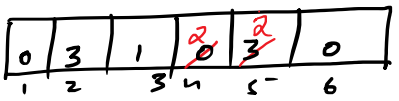
current = 3
visited[3] = 1, count = 2.

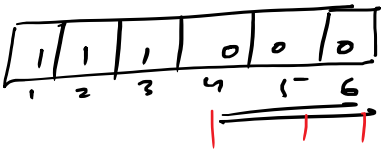


vertex no

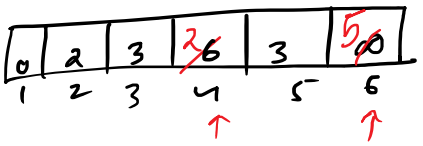


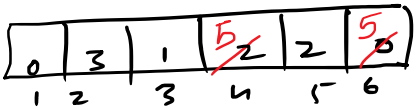
current = 2, visited[2] = 1, count = 3

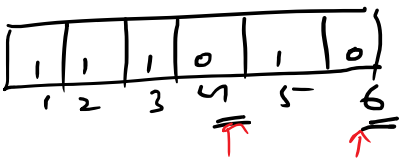




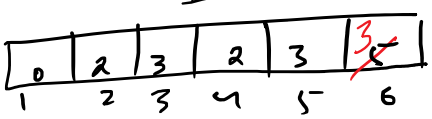
current = 5 visited[5] = 1 count = 4.

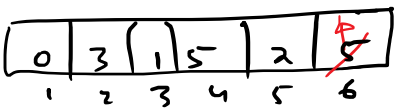


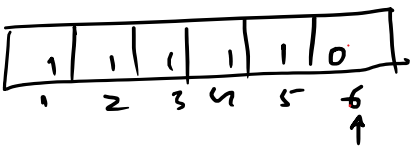




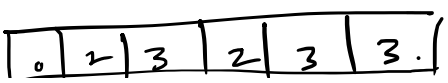
current = 4 visited[4] = 1 count = 5.

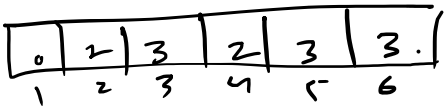


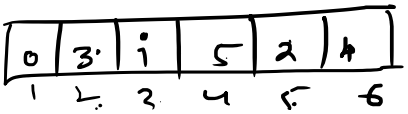


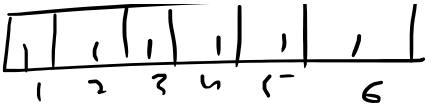


current = 6, visited[6] = 1 count = 6





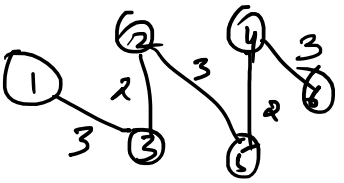




All the vertex visited.

Path from P[i] to i

1 to 3.
P[3] to 3
3 to 2
2 to 5
5 to 4
4 to 6.



MCST

Total cost = $\sum_{i=1}^{n} d[i] = 13$.

Prims Algo to find MST →

Comments: Let a graph be represented using adjacency matrix 'g'
Let there be 'V' no of vertices and 'e' number of edges.
Let there be 3- 1D array of V integers named as d, P and visited.

Here array
- d → stores cost
- p → stores path.
- visited stores state of vertex { visited = 1, unvisited = 0

Initialize array d to ∞
and array p & visited to 0.
Let 'c' be no of vertex visited, initialized c=0

```
Algo Prims (g, V, e)
{
  let current = 1;
  visited[current] = 1
  d[current] = 0
  c = 1;
  while (c != V) // All vertex are not visited
  {
    for (i = 1 to V) //chk with every other vertex
    {
      if( g[current][i] != 0  &&  visited[i] != 1 )
          (there is edge bet^n current & ith vertex)   (ith vertex is not already visited)
          (Edge hai)                                   (unvisited hai)
      {
        if ( g[current][i] < d[i] )
            (New cost from current to i)   (Earlier cost till i)
```

```
        if ( g[...] ...
        {  d[i] = g[current][i];  }
        } p[i] = current;  }
    }
}
```

current = vertex with smallest cost amongst unvisited vertices.

```
visited[current]=1;
c=c+1;
}// while
```

Total Cost = $\sum_{i=1}^{V} d[i]$

path for(i=2 to V)

path to 'i' from 'p[i]'

Dijkstras Algo to find Shortest Path from [Source to destination]

Comments: Let a graph be represented using adjacency matrix 'g'
Let there be 'V' no of vertices and 'e' number of edges.
Let there be 3- 1D array of V integers named as d, P and visited.

Here array
- d → stores cost
- p → stores path.
- visited stores state of vertex { visited = 1, unvisited = 0

Initialize array d to ∞
and array p & visited to 0.

Let src represent source/starting vertex and dest represents destination vertex
Let d_c = cost till current vertex.

Algo Prims (g, V, e)

{ let current = source;
visited[current] = 1
d[current] = 0
dc = 0;
while (current != dest) // untill dest is found.

{
for (i = 1 to V) //chk with every other vertex
{
if (g[current][i] != 0 && visited[i] != 1)
(there is edge bet^n current & i^th vertex — Edge hai; i^th vertex is not already visited — unvisited hai)
{
if (g[current][i] + dc < d[i])
(New cost from current to i — Earlier cost till i)

{ d[i] = g[current][i] + dc;

} p[i] = current;

}

}

current = vertex with smallest cost amongst unvisited vertices.

visited[current] = 1;

}// while

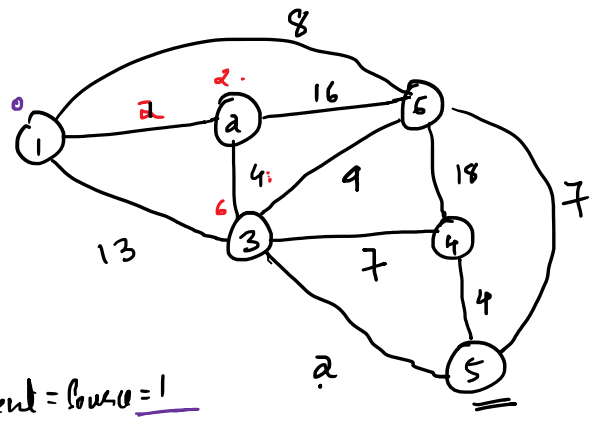
Cost of shortest path from src to dest = d[dest]

Print path for(i=2 to V)

path to 'i' from 'p[i]'

Print Path from src to dest via " p[dest]"

Single Source Shortest Path → (from Source to Destination)



Find shortest path

from Vertex 1 to 5
(source) (dest).

Using Dijkstra's Algorithm

current = source = 1

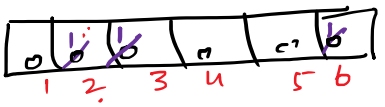
d[current] = 0

visited[current] = 1

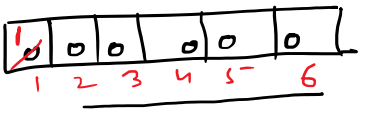
dc = cost till current = 0

d

P



visited

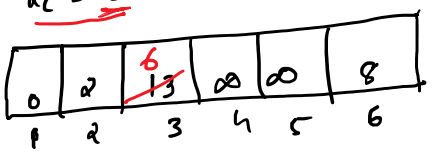


current = 2

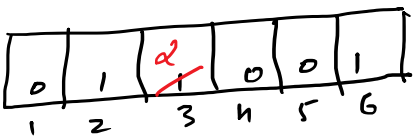
visited[2] = 1

dc = 2

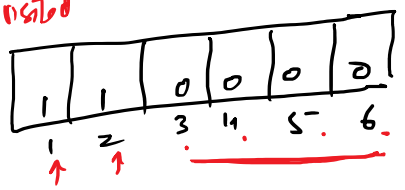
d



P



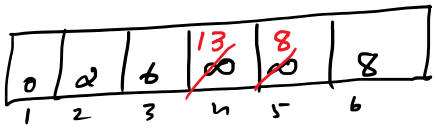
visited

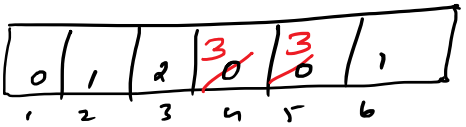


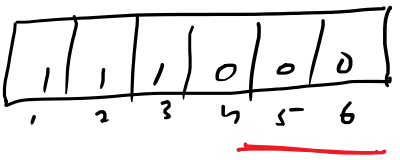
current = 3

visited[3] = 1

dc = 6

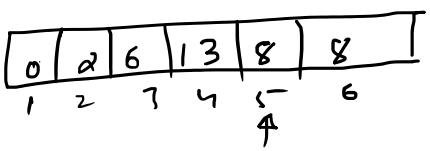


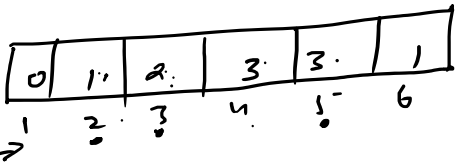


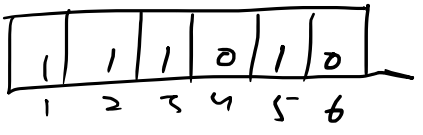


current = 5

visited[5] = 1





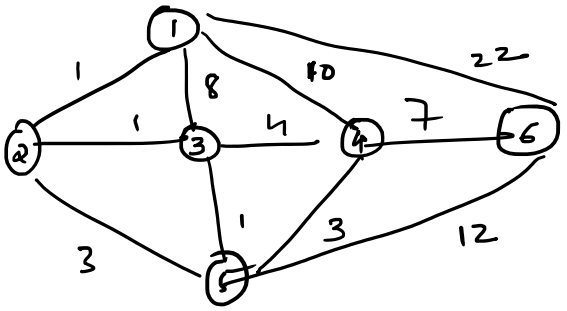


Since current = dist[5] = 5 Stop.

Cost from 1 to 5 = d[5] = 8

Path is ⇒. $1 \rightarrow 2 - 3 - 5$

H/W



Dijkstras Algo to find Shortest Path from [Source to All other Vertex]

Comments: Let a graph be represented using adjacency matrix 'g'
Let there be 'V' no of vertices and 'e' number of edges.
Let there be 3- 1D array of V integers named as d, P and visited.

Here array
- d → stores cost
- p → stores path.
- visited stores state of vertex { visited = 1, unvisited = 0

Initialize array d to ∞
and array p & visited to 0.

Let src represent source/starting vertex, Let c = No of vertex visited initialized = 0

Let d_c = cost till current vertex.

Algo Prims (g, V, e)

{ let current = source;
visited[current] = 1
d[current] = 0
d_c = 0; c = 1
while(c != V)) // untill dist is found.

{
for (i = 1 to V) //chk with every other vertex
{
if(g[current][i] != 0 && visited[i] != 1)
(there is edge bet^n current & i^th vertex — Edge hai; i^th vertex is not already visited — unvisited hai)

{
if (g[current][i] + d_c < d[i])
(New cost from current to i — Earlier cost till i)

{ d[i] = g[current][i] + dc;

p[i] = current;

}

}

current = vertex with smallest cost amongst unvisited vertices.

visited[current] = 1;

c = c + 1;

} // while

Cost of shortest path from src to dest = d[dest]

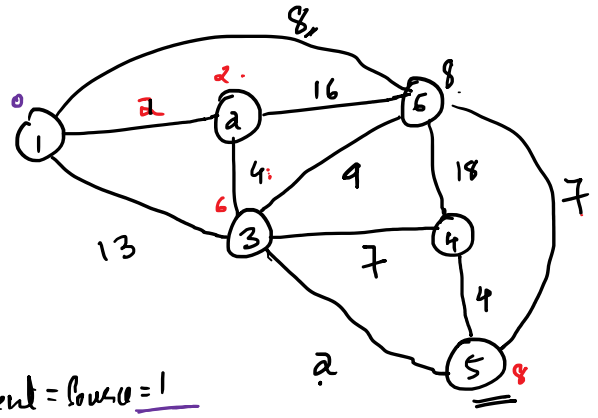
path for(i = 2 to V)

path to 'i' from 'p[i]'

for (i = 2 ; i < V; i++).

Path from 1 to 'i' via "p[i]"

Single Source Shortest Path → (from Source to All other Vertex).



Find shortest path

from Vertex 1 to All other Vertex.
(source)

Using Dijkstra's Algorithm

current = source = 1

d[current] = 0

visited[current] = C

d_c = cost till current = 0, c=1

d

1	2	3	4	5	6
0	2	13	∞	∞	8

P

1	2	3	4	5	6
0	1	1	0	0	0

visited

1	2	3	4	5	6
1	0	0	0	0	0

current = 2

visited[2] = 1

d_c = 2, c = 2

d

1	2	3	4	5	6
0	2	6	∞	∞	8

P

1	2	3	4	5	6
0	1	2	0	0	1

visited

1	2	3	4	5	6
1	1	0	0	0	0

current = 3

visited[3] = 1

d_c = 6, c = 3

1	2	3	4	5	6
0	2	6	13	8	8

1	2	3	4	5	6
0	1	2	3	3	1

1	2	3	4	5	6
1	1	1	0	0	0

current = 5

d_c = 8

visited[5] = 1, c = 4

1	2	3	4	5	6
0	2	6	13	8	8

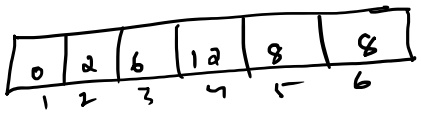
1	2	3	4	5	6
0	1	2	5	3	1

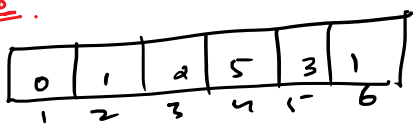
1	2	3	4	5	6
1	1	1	0	1	0

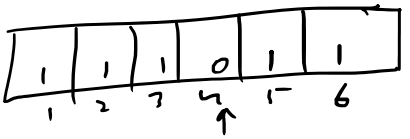
current = 6

visited[6] = 1, c = 5, d_c = 8.

visited[6] = 1, c = 5, $d_c = 8$.

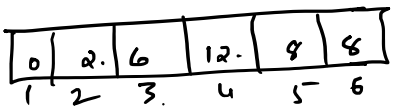


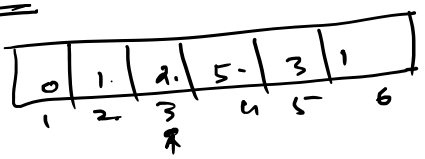




current = 4

visited[4] = 1, c = 6, $d_c = 12$





Now.

Shortest path cost from $1 \rightarrow 2 = 2$

$1 \rightarrow 3 = 6$ via 2

$1 \rightarrow 4 = 12$ via 5

$1 \rightarrow 5 = 8$ via 3

$1 \rightarrow 6 = 8$ directly from 1.

Knapsack Problem →

We are given an empty sack of capacity 'm'.
We were given n diff objects and their weights.
Weight of object i is represented as w[i] and profit as p[i]

We want to fill the sack with total capacity 'm'
Such that profit always remains maximum.

Let x[i]=0 if ith object is not added in sack.
x[i]=1 if ith object is added in sack.

The problem is to Maximize $\sum_{i=1}^{n} p_i \times x_i$
(profit, object)

Subjected to $\sum_{i=1}^{n} w_i \times x_i \leq m$

Consider $m = 15$, $n = 7$

P_1	P_2	P_3	P_4	P_5	P_6	P_7
10	5	15	7	6	18	3

w_1	w_2	w_3	w_4	w_5	w_6	w_7
2	3	5	7	1	4	1

Type 1) Fractional Knapsack.

Step 1 → Find P/W of each object [Per Unit Cost].

object	Profit	Weight	P/W
1	10	2	5
2	5	3	1.67
3	15	5	3
4	7	7	1
5	6	1	6
6	18	4	4.5
7	3	1	3

Step 2) Arrange the objects in Descending order of P/W

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

3	15	5	3
7	3	1	3
2	5	3	1.67
4	7	7	1

Step 3) capacity = m = 15

Object Added	Weight	Remaining capacity	Profit Earned
5	1	14	6
1	2	12	6+10=16
6	4	8	16+18=34
3	5	3	34+15 =49
7	1	2	49+3=52
2	2	0	52 + 1.67*2

= 55.34 ✓

Total profit = 55.34.

In Fractional Knapsack we are allowed to take fraction of object that will satisfy the capacity.

0/1 Knapsack → Here fraction of object is not Allowed - The object has to be Completely taken or rejected.

Consider $m = 15$, $n = 7$

P_1	P_2	P_3	P_4	P_5	P_6	P_7
10	5	15	7	6	18	3

w_1	w_2	w_3	w_4	w_5	w_6	w_7
2	3	5	7	1	4	1

Type2) 0/1 Knapsack.

Step1 → Find P/W of each object [Per Unit Cost].

Object	Profit	Weight	P/W
1	10	2	5
2	5	3	1.67
3	15	5	3
4	7	7	1
5	6	1	6
6	18	4	4.5
7	3	1	3.

Step2) Arrange the objects in Descending order of P/W

	Object	Profit	weight	P/W
1 →	5	6	1	6
2 →	1	10	2	5
3	6	18	4.	4.5
4	3	15	5	3
5	7	3	1	3
6	2	5	3.	1.67
7	4	7	7.	1

14 - 2 ≥ 0 Yes
C = 12
P = 6 + 10 = 16
12 - 4 ≥ 0 Yes.
C = 8

Step 3) capacity = m = 15

Object	weight w[i]	C Remaining	Profit

Profit

C - w[i]
2 - 3 ≥ 0 NO

Step 3) capacity =

Object Added	Weight (wt[i])	Remaining capacity (C)	Profit Earned
5	1	14	6
1	2	12	6+10=16
6	4	8	16+18=34
3	5	3	34+15=49
7	1	2	49+3=52

C - wt[i]
2 - 3 >= 0 NO

C = 2

Here remaining capacity is 2

but weight of object (Remaining object) > 2

So we will reject Both the object

∴ Total profit = 52.

2 capacity is not utilized.

Greedy Algo for Knapsack Problem →

Given an empty knapsack of capacity 'm' and n diff objects. Weights and profit for all n objects are stored in array w & p.

We want to fill the knapsack with entire capacity m such that the profit earned is maximum.

Algo Knapsack(m, n, w[], p[])

```
{  // Sort the p & w in decreasing order of p/w
   let c = m;
   profit = 0;
   // Fill the sack
   for (i=1 to n) // arranged in decending order of p/w
   {
      if (c - w[i] >= 0)   // c: remaining capacity
      {
         c = c - w[i];
         profit = profit + p[i];
      }
      Display object added with weight w[i]
       & profit p[i].
   }
   else
      break;
}//
```

```
if (i<=n)                                   52 +        1.67 ≈ 2
{   profit = profit + [(p[i]/w[i]) * c;
                          1.67                          2
    Display objed with weight c and
        profit earned (p[i]/w[i]) * c;
}
display maxprofit = Profit;
}
```

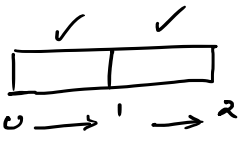
Greedy Approach →

Job Sequencing with Deadline →.

* Given n different jobs to be performed using Single machine.
* Profit and deadline for each job is specified.
* One job can be performed at a time.
* Every job requires one unit of time.
* We want to find subset of the given job such that all the jobs in the subset are completed within their deadline and profit earned is maximum.

Ex

job	1	2	3	4
profit	100	15	10	27
deadline	2	1	2	1



Possible Sol → ① $\langle 1, 3 \rangle$ ⇒ 100 + 10 = 110
(0-1, 1-2)

② $\langle 2, 1 \rangle$ ⇒ 15 + 100 = 115
(0-1, 1-2)

Sol → Step1 : Arrange the jobs in descending order of profit.

job	1	4	2	3
Profit	100	27	15	10

Job	1	4	2	3
Profit	100	27	15	10
deadline	2	1	1	2

Step2 → ① Add job1 to the sol →

J = { 1 } → done ⇒ profit earned = 100.
0-1

② Add job4 to the sol.

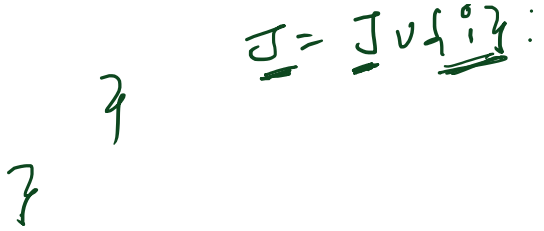
Since deadline of job4 is 1
and job1 is 2
Switch the order of Ex.

J = J ∪ {i} { 4, 1 } ⇒ 27 + 100 = 127
job4 0-1 1-2

Given the time limit we cannot add other job.

High level Description.

Algo JSD(d, J, n)
// J → set of jobs that can be completed by deadline.
J_1 = { 1 } // job with highest Profit
for (i = 2 to n) do // for all job from 2 ton.
if all job in J ∪ {i} can be completed
by deadline then



Compare Divide & Conquer & Greedy

Divide & Conquer.	Greedy.
(1) To obtain solution of given problem.	(1) used to obtain optimal solution.
(2) Problem is divided into Sub problems and soln of Subproblems are combined together to get sol of problem.	(2) Set of feasible soln is generated and optimal is selected.
(3) Here Duplicate soln may be obtained.	(3) The optimal is selected without revisiting previous sol.
(4) Less efficient as there is rework on soln	(4) More efficient than Divide & Conquer but it cannot guarantee optimal soln
(5) Ex Binary Search Merge Sort	(5) Ex: Knapsack Finding MST.

Q) Solve the problem of Job Sequencing with Deadline.

Job	Profit	Deadline
1	100	2
2	15	1
3	27	2
4	10	1

Sol ⇒

jobNo

job3	job1
1	2

Flag

∅ 1	∅ -1
1	2

S

	jobno	flag
1	job3	∅ 1
2	job1	∅ 1

Step 1) Consider jobs in Descending order of profit.

Consider job1

d(job1) = 2

check slot(2)

S(2).jobno = job1

S2.flag = 1

Profit = 0+100 = 100

Consider job3

d(job3) = 2

check slot 2 ⇒ Not Empty.

chk for slot 1 = Empty (return)

S(1).jobNo = job3

S(1).flag = 1

Profit = 100+27 = 127

Consider job 2

d(job 2) = 1

chk slot 1 = not Empty

job2 cannot be scheduled

consider job4

d(job4) = 1

chk slot 1 = not Empty

job4 cannot be scheduled.

Profit = 127
Schedule = { Job3, Job1 }

Q) $(P_1 \ldots P_7) = (3, 5, 20, 18, 1, 6, 30)$
$(d_1 \ldots d_7) = (1, 3, 4, 3, 2, 1, 2)$

} Solve Job Sequencing Deadline with Greedy Approach.

Sol	Job	P	d
	1	3	1
	2	5	3
	3	20	4
	4	18	3
	5	1	2
	6	6	1
	7	30	2

S (slot)	i ✓ jobno	✓ flag
0		
→ 1	job 6	✓ 1
2	job 7	✓ 1
3	job 4	✓ 1
4	job 3	✓ 1

Highest deadline = 4 = 4 slots

① Consider job7.
d(job7) = 2
chk slot 2 ⇒ Empty.
s(2).jobno = 7
s(2).flag = 1
Profit = 30

② Consider job3
d(job3) = 4
check slot 4 = Empty.
s(4).jobno = job3
s(4).flag = 1
Profit = 30 + 20 = 50

Consider job 4
d(job4) = 3
chk slot 3 = Empty.
s(3).jobno = job4
s(3).flag = 1
Profit = 50 + 18 = 68.

④ Consider job6
d(job6) = 1
chk slot 1 = Empty.
s(1).jobno = job6
s(1).flag = 1
Profit = 68 + 6 = 74

Now Since all the slots are occupied.
So no other jobs can be scheduled.

Total prof.t = 74.

Schedule → job6 → job7 → job4 → job3

Algo for Job Sequencing with Deadline.

function GetSlot (S, dl) : Integer
{ // will send the slot deadline and will return whether the slot is full or not

S(1:k) : Here the schedule contains job no and flag.

dl => deadline.

```
1. start
2. for ( i = dl to 1 )
3. { if (S(i).flag == 0)
       return i
   }
4. return -1;
}//
```

Algo for function Job Sequencing (X, n, S, K)

{ where

X(1:n) : Input array containing x.j (job no), x.p (profit)-
and x.d (deadline).

n = no of slots available (Equal to largest deadline value).

S(1:k) = Schedule contains jobno and flag
for k scheduled job out of n.

```
1. start
2. profit = 0
```

3. Sort all the jobs in descending order of profit.
4. for(i=1 to n) ⇒ i=1 job with highest profit.
5. { avail = GetSlot(s, x(i).d)
6. if (avail == -1) then
7. print "job" x(i).j "cannot be Scheduled"
else
8. { s(avail).jobno = x(i).j;
9. s(avail).flag = 1
10. profit = profit + x(i).p.
}
}
11. Print S (schedule).
12. Print Profit
13. Return.

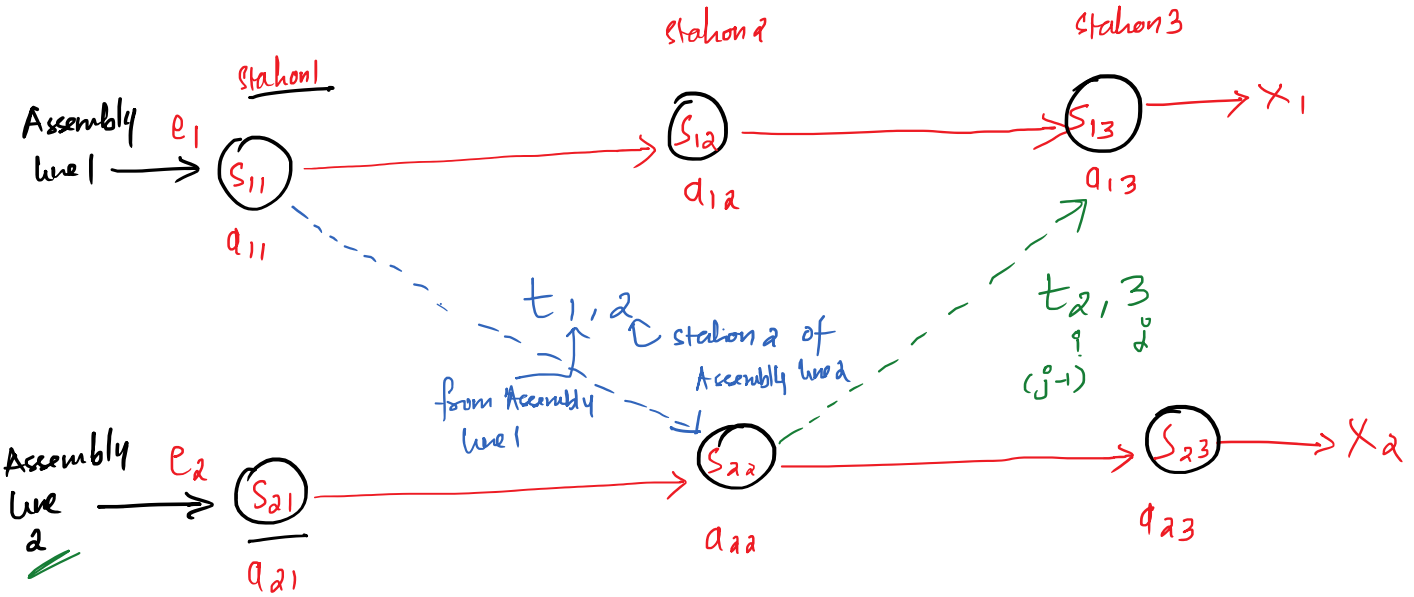
Dynamic Programming

Assembly line Scheduling →

* A factory has 2 assembly line with n stations each.
* A station is denoted as $S_{i,j}$ (i: assembly line no, j: station no)

i = Assembly line no

j = station no.



$t_{i,j}$ ⇒ Time required to switch from station $(j-1)$ ie 1 of Assembly line i (1) to station j (2) of other Assembly line. (1+1=2)

Here $a_{i,j}$ = Time taken on Assembly line i & station j.

I must pass through each of n stations in order

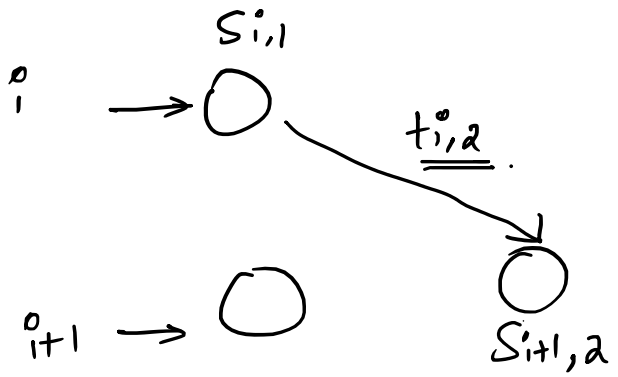
* Product must pass through each of n stations in order before Exiting the Assembly line.

* The parallel station of the two assembly line performs the same task.

* After product passes through $S_{i,j}$ then would continue to $S_{i,j+1}$ unless it decides to shift to other Assembly line.

* Continuing on same assembly line takes no extra cost.

* But transferring from assembly line i at station $j-1$ to station j, on other assembly line will take t_{ij}

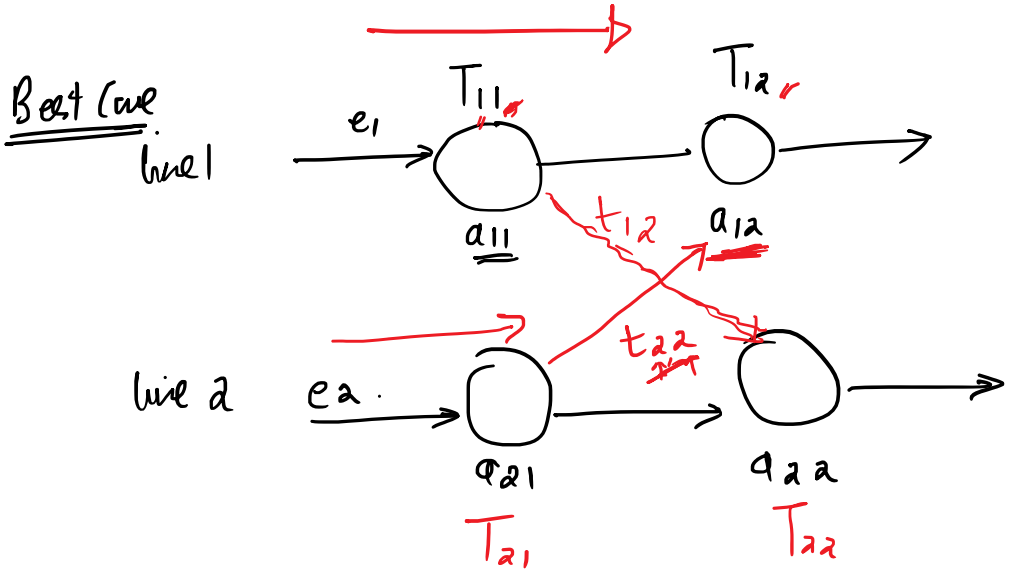


* Each assembly line takes an entry time e_i
* Each assembly line takes an exit time x_i
* Entry and Exit Time may be diff for diff assembly line.

Approach => The decision involves determining whether the assembly line should be switched or not.

$T_{1,j}$ => Indicates min time taken to leave station j on assembly line 1.

$T_{2,j}$ => " " " " " " " j on assembly line 2.



Total time to leave station 1.

$T_{11} = e_1 + a_{11}$ ← Entry time in assembly line + Time spent on station 1.

$T_{21} = e_2 + a_{21}$

$T_{12} = \min$ { $T_{11} + a_{12}$ } Comming from station 1 of line 1

T_{11} ↑ Time taken to leave station 1 of line 1

a_{12} ↑ Time spent on st^n 2 line 1

{ $T_{21} + t_{2,2} + a_{12}$

T_{21} ↑ Time to

$t_{2,2}$ ↑

a_{12} ↑ Time spent on st^n 2 of line 1

leave stn 1 of line 2

time to switch from line 2 stn 1 to line 1 stn 2

$$T_{22} = \min \begin{cases} T_{21} + a_{22} & \text{comming from line 2} \\ T_{11} + t_{12} + a_{22} & \text{comming from line 1} \end{cases}$$

Total Time taken to Come out of factory →. There are n station

$$T^{*}_{min} = \min \begin{cases} \underbrace{T_{1,n} + x_1}_{\text{Exit time of station n Assembly 1}} & , \quad \underbrace{T_{2n} + x_2}_{\text{Exit time of station n Assembly 2.}} \end{cases}$$

Assembly line 1 | Assembly line 2

Problem $n = 4$ (no of stations)

$a(2,4) = \{ \{4, 5, 3, 2\}, \{2, 10, 1, 4\} \}$

$a_{11}\ a_{12}\ a_{13}\ a_{14}\quad a_{21}\ a_{22}\ a_{23}\ a_{24}.$

$t(2,4) = \{ (0, 7, 4, 5), (0, 9, 2, 8) \}$

$e(1,2) = \{10, 12\}$ (e_1, e_2)

$x(1,2) = \{18, 7\}$ (x_1, x_2)

Sol →

a	1	2	3	4
1	4	5	3	2
2	2	10	1	4

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

$t_{23} = 2$

time spend to leave line 2 st^n 2 to reach st^n 3 of line 1

Sol $j = 1$ (station 1).

$T_{11} = e_1 + a_{11} \Rightarrow 10 + 4 = 14$

$T_{21} \Rightarrow e_2 + a_{21} \Rightarrow 12 + 2 = 14.$

$j = 2$ (station 2).

$T_{12} = \min(T_{11} + a_{12}, T_{21} + t_{22} + a_{12})$

$$T_{1a} = \min\left(T_{11} + a_{1a} \;,\; T_{21} + t_{22} + a_{12} \right)$$

$$= \min\left(14 + 5 \;,\; 14 + 4 + 5 \right)$$

$$= \min(19, 28)$$

$= 19$. [When st^n 2 of line 1 is leaving it is Comming from st^n 1 of line 1 only]

$$T_{2a} = \min\left(T_{21} + a_{22} \;,\; T_{11} + t_{12} + a_{22} \right)$$

$$= \min(14 + 10, 14 + 7 + 10)$$

$$= \min(24, 31)$$

$= 24$ [When st^n 2 of line 2 is leaving, it is Comming from st^n 1 of line 2]

$i = 3$ (Station 3)

$$T_{13} = \min\left(T_{1a} + a_{13} \;,\; T_{22} + t_{13} + a_{13} \right)$$

$$= \min(19 + 3, 24 + 2 + 3)$$

$$= \min(22, 29)$$

$= 22$

$$T_{23} = \min\left(T_{2a} + a_{23} \;,\; T_{1a} + t_{13} + a_{23} \right)$$

T_{23} = min ($t_{22} + a_{23}$, $t_{12} + t_{13} + a_{23}$)

= min (24+1 , 19+4+1)

= min (25 , 24)

= 24 [Leaving stn 3 of line 2 ; it is switching from stn 2 of line 1].

j = 4 (Skelwonk)

T_{14} = min ($T_{13} + a_{14}$, $T_{23} + t_{24} + a_{14}$)

= min (22+2 , 24+8+2)

= min (24 , 34)

= 24

T_{24} = min ($T_{23} + a_{24}$, $T_{13} + t_{14} + a_{24}$)

= min (24+4 , 22+5+4)

= min (28 , 31)

= 28

T_{min} = min ($T_{14} + X_1$, $T_{24} + X_2$)

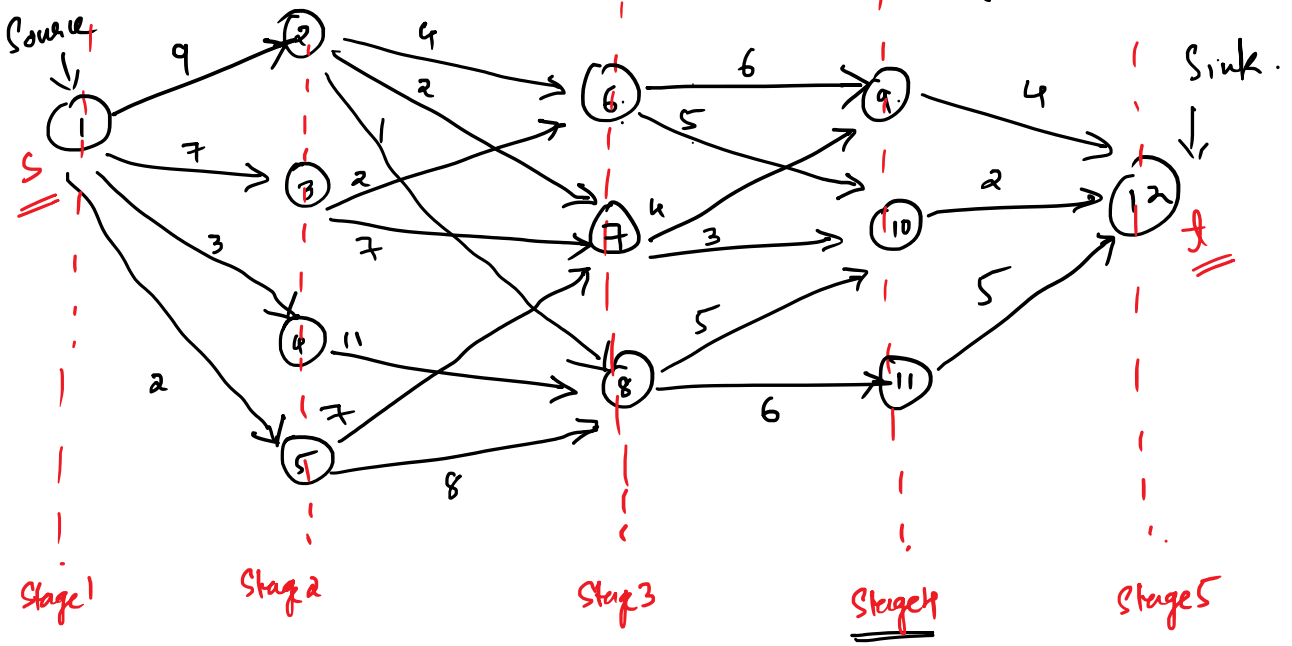
= min (24+18 , 28+7)

$= \min(42, 35)$

$= 35$ //

Multistage Graph → Using Dynamic Programming →

Example → Shortest path from Source (s) to Sink (t)



Observations → (1) All the edges are Directed

(2) The edge has some direction from Source toward Sink.

(3) The Edges are from $i-1^{th}$ stage to i^{th} stage

(4) We divide the overall graph into diff stages and collect the nodes in form of stages.

Ex In stage 1 = {1}

Stage 2 = {2,3,4,5}

Stage 3 = {6,7,8}

Stage 4 = {9,10,11}

Stage 5 = d12?

Sol^{n} → We will start from Sink and this approach is known as forward approach.

Step1 → Stage 5

$Cost(5, 12) = 0$

(5: Stage NO, 12: node NO)

[Stage 5 pe node 12 se aage ka Cost]

Step2: Stage 4

$cost(4, 9) = C_{9,12} = 4$, $D(4,9) = 12$

(4: Stage, 9: node no, $C_{9,12}$: cost of Edge 9-12, D(4,9): Decision take to go to which node from node 9 at stage 4)

$cost(4,10) = C_{10,12} = 2$, $D(4,10) = 12$

$cost(4,11) = C_{11,12} = 5$, $D(4,11) = 12$

Stage 3

$$Cost(3,6) = \min \{ C_{6,9} + Cost(4,9) \text{ (edge 6 to 9)} , C_{6,10} + Cost(4,10) \}$$

$$= \min \{ 6 + 4 , 5 + 2 \}$$

$= 7$ //

$\boxed{D(3,6) = 10}$

$cost(3,7) = \min\{ c_{7,9} + cost(4,9) , c_{7,10} + cost(4,10) \}$

$= \min\{ 4+4 , 3+2 \}$

$= 5$ //

$\boxed{D(3,7) = 10}$

$cost(3,8) = \min\{ c_{8,10} + cost(4,10) , c_{8,11} + cost(4,11) \}$

$= \min\{ 5+2 , 6+5 \}$

$= 7$

$\boxed{D(3,8) = 10}$

Stage 2

$cost(2,2) = \min\{ c_{2,6} + cost(3,6) , c_{2,7} + cost(3,7) , c_{2,8} + cost(3,8) \}$

$= \min\{ 4+7 , 2+5 , 1+7 \}$

$= 7$

$\boxed{D(2,2) = 7}$

$cost(2,3) = \min\{ c_{3,6} + cost(3,6) , c_{3,7} + cost(3,7) \}$

$= \min\{ 2+7, 7+5 \}$

$= 9$

$\boxed{D(2,3) = 6}$

$cost(2,4) = \min\{ c_{4,8} + cost(3,8) \}$

$= 11+7 = 18$

$\boxed{D(2,4) = 8}$

$cost(2,5) = \min\{ (c_{5,7} + cost(3,7)), (c_{5,8} + cost(3,8)) \}$

$= \min\{ 7+5, 8+7 \}$

$= 12$

$\boxed{D(2,5) = 7}$

Stage 1 ->.

$cost(1,1) = \min\{ c_{1,2} + cost(2,2), c_{1,3} + cost(2,3), c_{1,4} + cost(2,4), c_{1,5} + cost(2,5) \}$

$= \min\{ 9+7, 7+9, 3+18, 2+12 \}$

$= \min\{ 16, 16, 21, 14 \}$

$= 14$

$\boxed{D(1,1) = 5}$

Path ->

D(1,1) ⇒ 5
↑ stage 2

D(2,5) ⇒ 7
stage 3

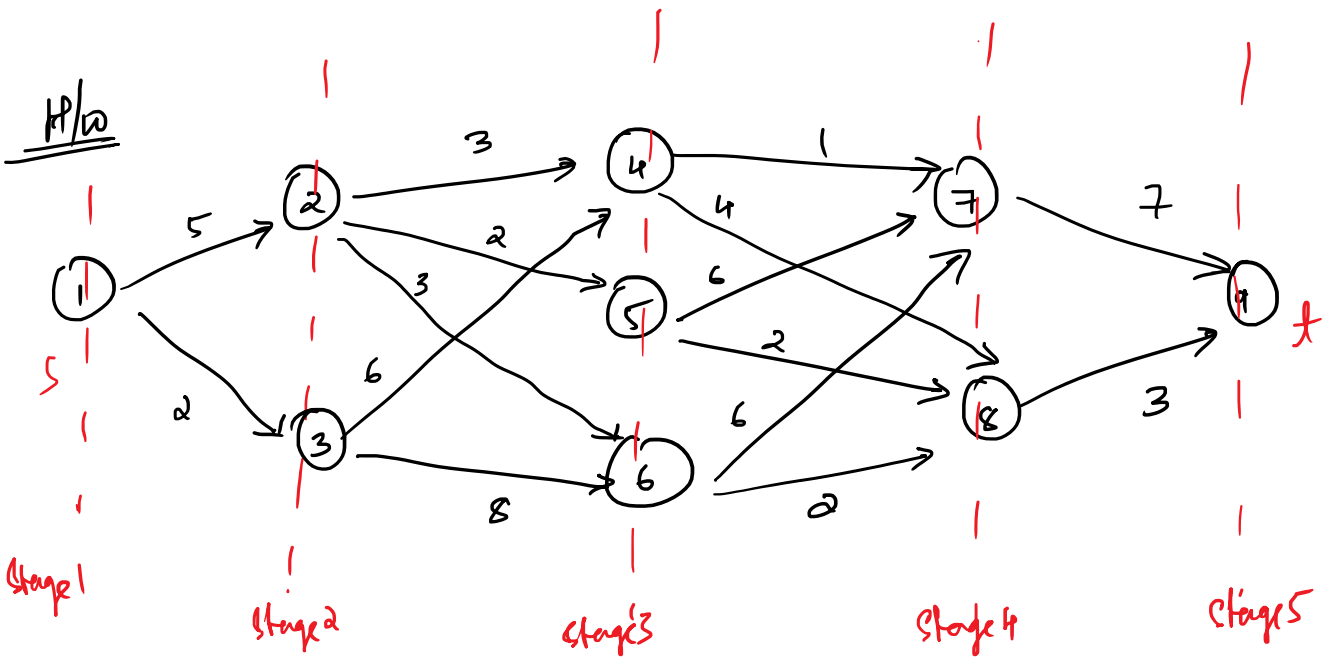
D(3,7) = 10
stage 4

D(4,10) = 12.

Path 1 → 5 → 7 → 10 → 12

Cost = 16.

H/W



Multistage Graph → It is directed graph where the vertices are partitioned into K disjoints subsets $K \geq 2$
(stages)

Stages are represented as $V_1, V_2, V_3 \ldots V_K$

If edge $(u,v) \in E(G)$

then if $u \in V_i$ [i^{th} stage]
then $v \in V_{i+1}$ [$i+1^{th}$ stage (next stage)].

Here V_1 is Source
V_K is Sink.

Here Cost of Path from Source (s) to Sink (t) is Sum of cost of all the edges on path from s to t.

Approach. → Here we implement forward approach where the decision involves vertex from next stage to be taken along the path.

Formula $cost(i,j) = \min \{ C_{j,l} + cost(i+1, l) \}$

- i → stage no
- j → vertex no
- $C_{j,l}$ → cost of edge from vertex j to some vertex l of stage $i+1$
- $cost(i+1, l)$ → further cost from vertex l of stage $i+1$ to Sink.

Here $l \in V_{i+1}$
one vertex of next stage.

any vertex of next stage.

$D(i,j)$ = Decision taken on stage i at vertex j to go to some vertex l in stage $i+1$ of Minimum Cost

Algo → function M_stage Graph(C, k, n, p)

{ Here C = cost matrix (adjacency matrix with cost
k = no of stages
n = no of vertex
P = path array.

Integer Const : cost[1:n], D[1:n], j, r, min.

1. Start
2. cost(n) = 0
3. for j = n-1 to 1 (second last → first)

{ min = ∞
for (r = 1 to n) do

{ if (c(j, r) ≠ 0 && c(j, r) + cost(r) < min)
(Edge Hai)

{ min = c(j, r) + cost(r);
Pmin = r;

}
cost(j) = min;
D(j) = Pmin.

}

}

}

P(1) = 1 ← vertex } first stage vertex 1

P(k) = n } k^{th} stage vertex n.

for (j = 2 to k-1) } stage 2 to k-1

P(j) = D(P(j-1))

return ;

}.

Longest Common Subsequence →

Consider $X = A\ B\ C\ B\ D\ A\ B$

$Y = B\ D\ C\ A\ B\ A$

Possible Common Subsequence →

A B B A — X ✓, Y ✗ } Not Common Subsequence.

B C A B — X ✓, Y ✓ } Common Subsequence.

There can be many possible Common Subsequence.

We want Longest Common Subsequence.

Appln

① Molecular Biology

A DNA sequence is represented as sequence of 4 letters A, C, G, T, using LCS we can find Common Subsequences to match DNA strands.

② LCS can also be used to determine 2 versions of same file by finding LCS of lines of 2 files.

Trick → if (X == Y)

{ then value = diag + 1

and arrow = ↖ (diag)

}

else if (upper ≥ left)

{ value = upperwale ka value.

arrow = ↑ (up)

}

else { value = leftwale ka value

arrow = ← left

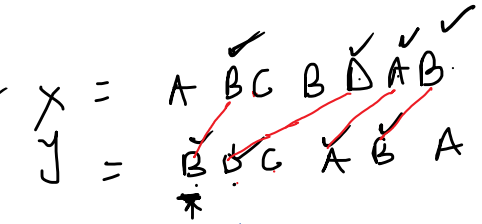
}

Longest Common Subsequence → (LCS)

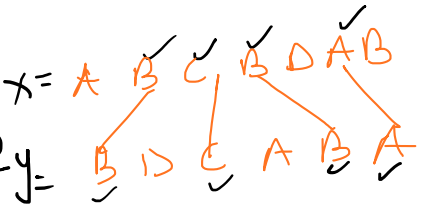
* We will solve it using Dynamic Programming

* LCS of 2 sequences is a subsequence with maximum length, which is Common to both the Sequence

Consider ✓ X = A B C B D A B
✓ Y = B D C A B A



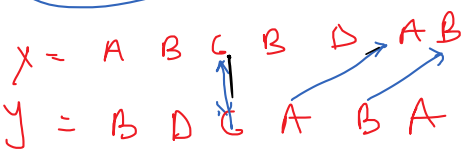
Also Y = B D C A B A



BCBA is also common Subsequence

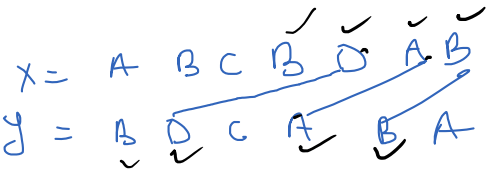
Try kaste B DAB. 4 is Common Subsequence

Also



CAB 3 is also Common Subsequence ✓

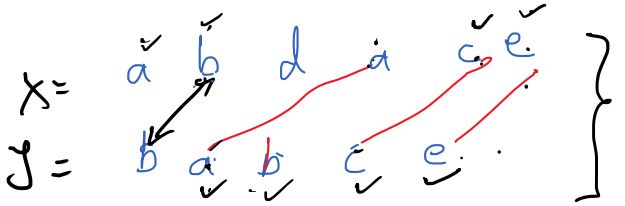
Also



DAB 3 is also Common Subsequence ✓

Longest Common Subsequence = 4 = BDAB ✓

Consider →



No Common Subsequence of length > 4

Common Subsequences ⇒ b a c e → 4
= a b c e → 4
a c e

Longest Common Subsequence

↘ (have more than one LCS for given pairs of Sequences

* Can have more than one LCS for given pair of Sequences

Appln =>

(1) Molecular Biology =>

A DNA Sequence is represented as sequence of 4 letters A C G T Using LCS we can find the common subsequences to match the DNA strands

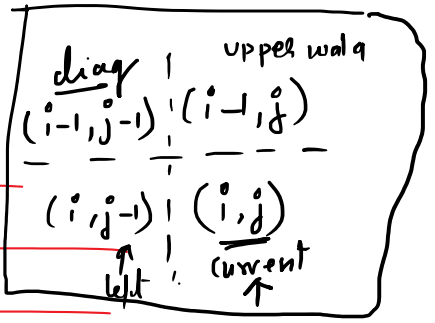
(2) LCS can also be used to determine 2 versions of same file by finding common subsequences of lines of 2 files.

Find LCS of the 2 Sequences Given using Dynamic Prog. Vimp

X = A B C B D A B = 7

Y = B D C A B A = 6

x ↓ / y →	y	0	1 B	2 D	3 C	4 A	5 B	6 A
0		0	0	0	0	0	0	0
1	A	0	0 ↑	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4



dir BC B A

trick if (x == y)

{ then value is diag + 1
and arrow ↖ (diag)
}

else if (upper wale ≥ left)

{ value = upperwale ka value
arrow ↑ (up)
}

else

{ value = leftwale ka value
arrow = ← (left)
}

else
{ value = leftwale ka value
arrow = ← (left)
}

Algorithm → Sequences.

LCS (X, Y)

1 m ← length (X) ✓
2. n ← length (Y) ✓
3 for (i = 0 to m) do } first row & first col to zero
// row
First col // c[i, 0] = 0 ✓
4 for (j = 0 to n) do
First Row. c[0, j] = 0

6 for (i = 1 to m) do — m
7 { for (j = 1 to n) do — n
8 { if (Xi == Yj) then } if same Hai
9 { c[i, j] = c[i-1, j-1] + 1 } diag+1 & ↖
10 b[i, j] = '↖'
}
11 else if (c[i-1, j] >= c[i, j-1])
upperwala leftwala
12 { c[i, j] = c[i-1, j]
13 } b[i, j] = '↑'
else
14 { c[i, j] = c[i, j-1]; leftwale

14 else c[i,j] = c[i,j-1]; leftwards

15. b[i,j] = '←'

}

}

}

16 return c & b array

To print solution =>

i = m

j = n

LCS(b, X, i, j)

if i = 0 or j = 0

return.

if b[i,j] = '↖'

LCS(b, X, i-1, j-1) } go diag

Print X → print entry.

else if b[i,j] = '↑'

LCS(b, X, i-1, j) → go upward

else //←

LCS(b, X, i, j-1) → go left.

(4) Travelling Salesman Problem

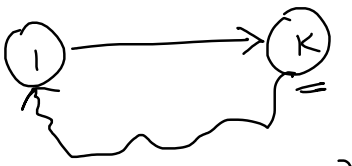
Let $G = (V, E)$ be a directed graph with edge cost C_{ij} such that

$C_{ij} > 0$ if $\langle i, j \rangle \in E(G)$.

$C_{ij} = \infty$ if $\langle i, j \rangle \notin E(G)$.

* A tour is a directed cycle that includes every vertex of graph G.
* The cost of the tour is the sum of cost of all the edges of tour
* The problem is to find the tour with minimum cost.

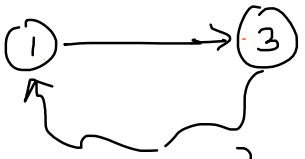
Approach :>



$V - \{1, K\}$

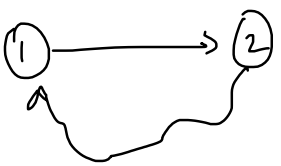
start from 1
Go to K
Then from K to 1
visiting all the vertex other than 1, K.

ex $n = 5$



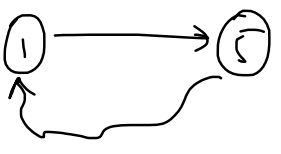
$\{2, 4, 5\}$

OR



$\{3, 4, 5\}$

OR



$\{2, 3, 4\}$

A tour is a simple path that starts and ends at same vertex.

"A graph tour consist of an edge $\langle 1, k \rangle$ for some $k \in V - \{1\}$ and a path from vertex k to vertex 1 which must go through all the vertex in $V - \{1, k\}$ exactly once.

$g(2, \emptyset)$

* let $g(i, S)$ be the length of the shortest path from vertex i to vertex 1 which must go through each vertex in set S exactly once

∴ Hence $g(1, V - \{1\})$ represent length of optimal graph tour

$g(1, \{2,3,4\})$

(start from 1 come back to 1 visiting all the vertex in $V - \{1\}$)

Generalizing → from vertex i go to vertex 1 visiting all the vertex in set S.

$$g(i, S) = \min_{j \in S} \left(c_{ij} + g(j, S - \{i, j\}) \right)$$

i start

$j \in S$

j se 1 via $S - \{i, j\}$

To start from $i \rightarrow$ go from $i \rightarrow j \Rightarrow c_{ij}$

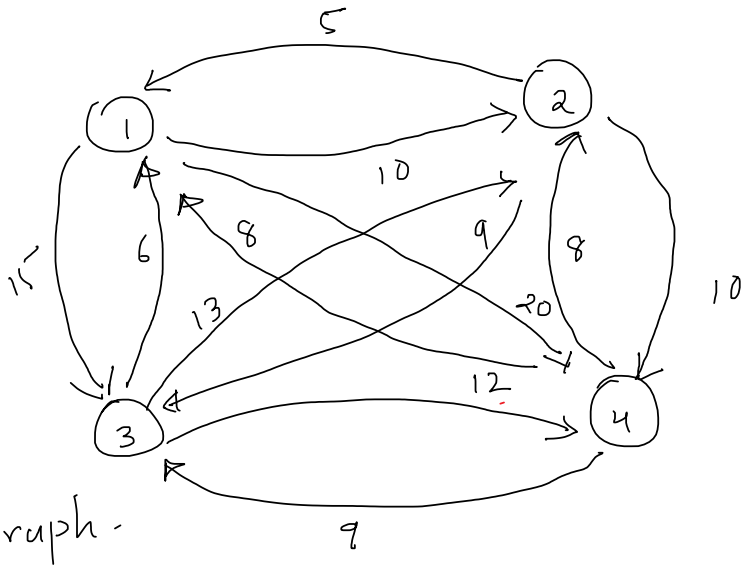
Then come back from $j \Rightarrow g(j, S - \{i, j\})$

to i visiting all
vertex V-{i, j}

We have $g(i, \phi) = \{C_{i1}\}$ $\quad C_i^0 \rightarrow 1$

from vertex i to 1
visiting no other nodes since ϕ

Question



Find Min Tour
Cost of Given Graph.

start tour from vertex = 1

Sol → $|s| = 0$

$g(2, \phi) = C_{21} = 5$
$g(3, \phi) = C_{31} = 6$
$g(4, \phi) = C_{41} = 8$

$|s| = 1$ → from 2 to 1 via 3 ⇒ 2→3 then 3→1

$g(2, \{3\}) = C_{23} + g(3, \phi) = 9 + 6 = 15$

$g(2, \{4\}) = C_{24} + g(4, \phi) = 10 + 8 = 18$

$= C_{32} + g(2, \phi) = 13 + 5 = 18$

From any vertex other than 1 comming

$g(1,\{2,3,4\})$

$g(3,\{2,4\}) = C_{32} + g(2,\emptyset) = 13+5 = 18$

$g(3,\{4\}) = C_{34} + g(4,\emptyset) = 12+8 = 20$

$g(4,\{2\}) = C_{42} + g(2,\emptyset) = 8+5 = 13$

$g(4,\{3\}) = C_{43} + g(3,\emptyset) = 9+6 = 15.$

$g(4,\{3\}) \Rightarrow 3$

than 1 comming back to one using only one intermediate Vertex

$|S| = 2$ → 2 sel via 3 & 4

2 se 3 then 3 to 1 via 4

2 se 4 then 4 to 1 via 3

$g(2,\{3,4\}) = \min$ $\min(C_{23} + g(3,\{4\}),\ C_{24} + g(4,\{3\}))$

$= \min(9+20,\ 10+15) = \min(29,25) = 25$

$d(2,\{3,4\}) = 4$

$g(3,\{2,4\}) = \min(C_{32} + g(2,\{4\}),\ C_{34} + g(4,\{2\}))$

$= \min(13+18,\ 12+13) = 25$

$d(3,\{2,4\}) \Rightarrow 4$

$g(4,\{2,3\})$ (4 is i, S is $\{2,3\}$)

4 se 2 and 2 se 1 via 3.

$S-\{i,j\}$

$\min(C_{4,2} + g(2,\{3\}),\ C_{43} + g(3,\{2\}))$

$= \min(8+15,\ 9+18) = 23.$

$d(4,\{2,3\}) = 2$

$|S| = 3$

$g(1,\{2,3,4\})$

1 sel via 2,3,4

$1 \to 2$ then 2 sel via 3,4

$1 \to 3$ then 3 sel via 2,4

$1 \to 4$ then 4 sel via 2,3.

$= \min\{ C_{12} + g(2,\{3,4\}),\ C_{13} + g(3,\{2,4\}),\ C_{14} + g(4,\{2,3\}) \}$

$= \min\{ 10 + 25,\ 15 + 25,\ 20 + 23 \}$

$= \min\{ 35, 40, 43 \}$

$= 35$

$d(1,\{2,3,4\}) \Rightarrow 2$

Total Tour Cost $= 35$

Path $d(1,\{2,3,4\}) = 2$ ✓ path

$d(2,\{3,4\}) \Rightarrow 4$ ✓

$d(4,\{3\}) = 3$ ✓

$d(3,\{\emptyset\}) \Rightarrow 1$

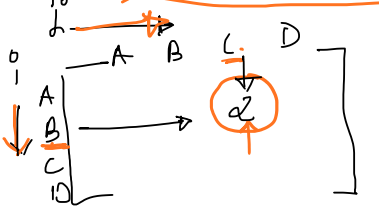
$1 \to 2 \to 4 \to 3 \to 1$

H/W

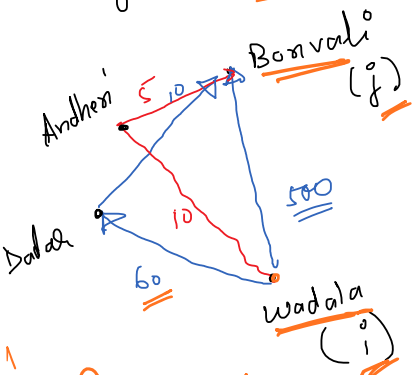
	1	2	3	4
1	0	15	20	3
2	5	0	18	6
3	4	8	0	12
4	8	2	18	0

All pair shortest Path →

The all pair shortest path problem is to determine a matrix C (cost matrix) such that c[i][j] is the length of shortest path from node i to node j



* The matrix $c^k[i][j]$ represents length of shortest path from i to j going through some vertex k.



c(wadala, Borivali)

= min (c[wadala][borivali] , c[wadala][dadar] + c[dadar][borivali] , c[wadala][andheri] + c[andheri][borivali])

=> min {500, 70, 15}

Floyd Warshalls Algo

```
for (k=1 ; k<=v ; k++)
{
    for (i=1 ; i<=v ; i++)
    {
        for (j=1 ; j<=v ; j++)
        {
            c[i][j] = min ( c[i][j] , c[i][k] + c[k][j] )
        }
    }
}
```

vertex pair i, j ⟹ $O(v^3)$

c[i][j] : direct path from i to j

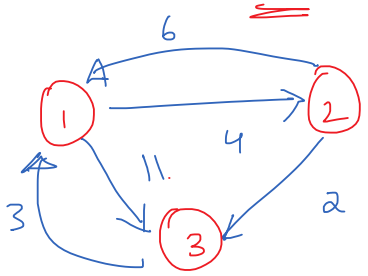
c[i][k] + c[k][j] : path from i to k and from k to j

k is intermediate vertex, k = 1 to v

=> we want shortest path length from every vertex i to every other vertex j

Find All pair Shortest Path in the given graph

Q) Given



step1:

$$C = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix} \end{array}$$

if there is no direct edge from i to j then $C[i][j] = \infty$

step2 $k=1$ [through 1]

$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 4 & 11 \\ 6 & 6 & 2 \\ 3 & \infty & 0 \\ & 7 & \end{bmatrix} \end{array}$$

[3,2] cost = 7 via 1

Decision

$$D = \begin{array}{c} \\ \rightarrow 1 \\ \rightarrow 2 \\ \rightarrow 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} -1 & 1 & 1 \\ 2 & -1 & 2 \\ 3 & 1 & -1 \end{bmatrix} \end{array}$$

from vertex 2 to 3 from 2 (no via)

D[i][j] = -1 if no edge from i to j

from 3 to 2 via 1

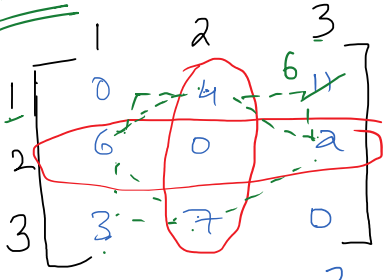
$$C[2,2] = \min(C[2,2],\ C[2][1] + C[1][2])$$
$$= \min(0,\ 6+4) = \min(0,10)$$
$$= 0$$

understanding

$$C = A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix} \end{array}$$

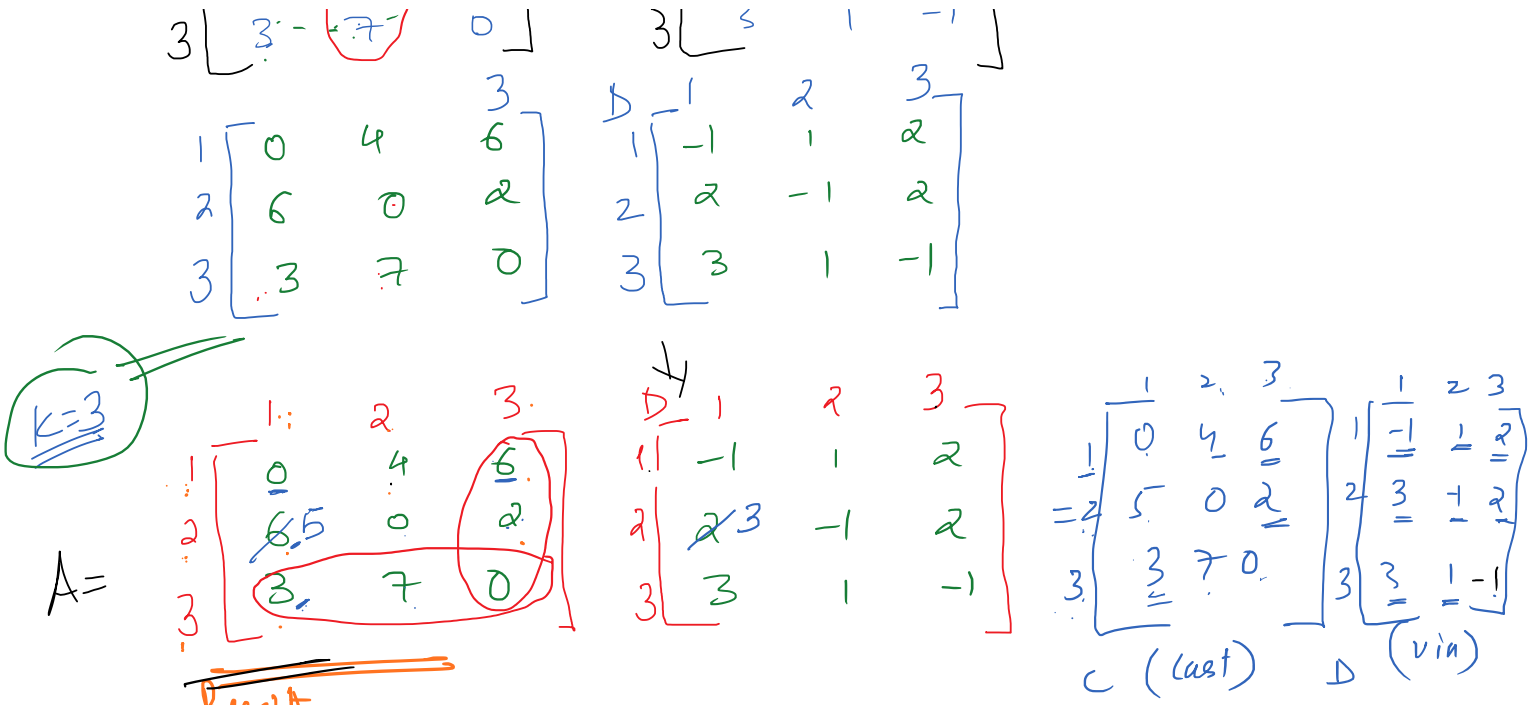
$$D = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} -1 & 1 & 1 \\ 2 & -1 & 2 \\ 3 & 1 & -1 \end{bmatrix} \end{array}$$

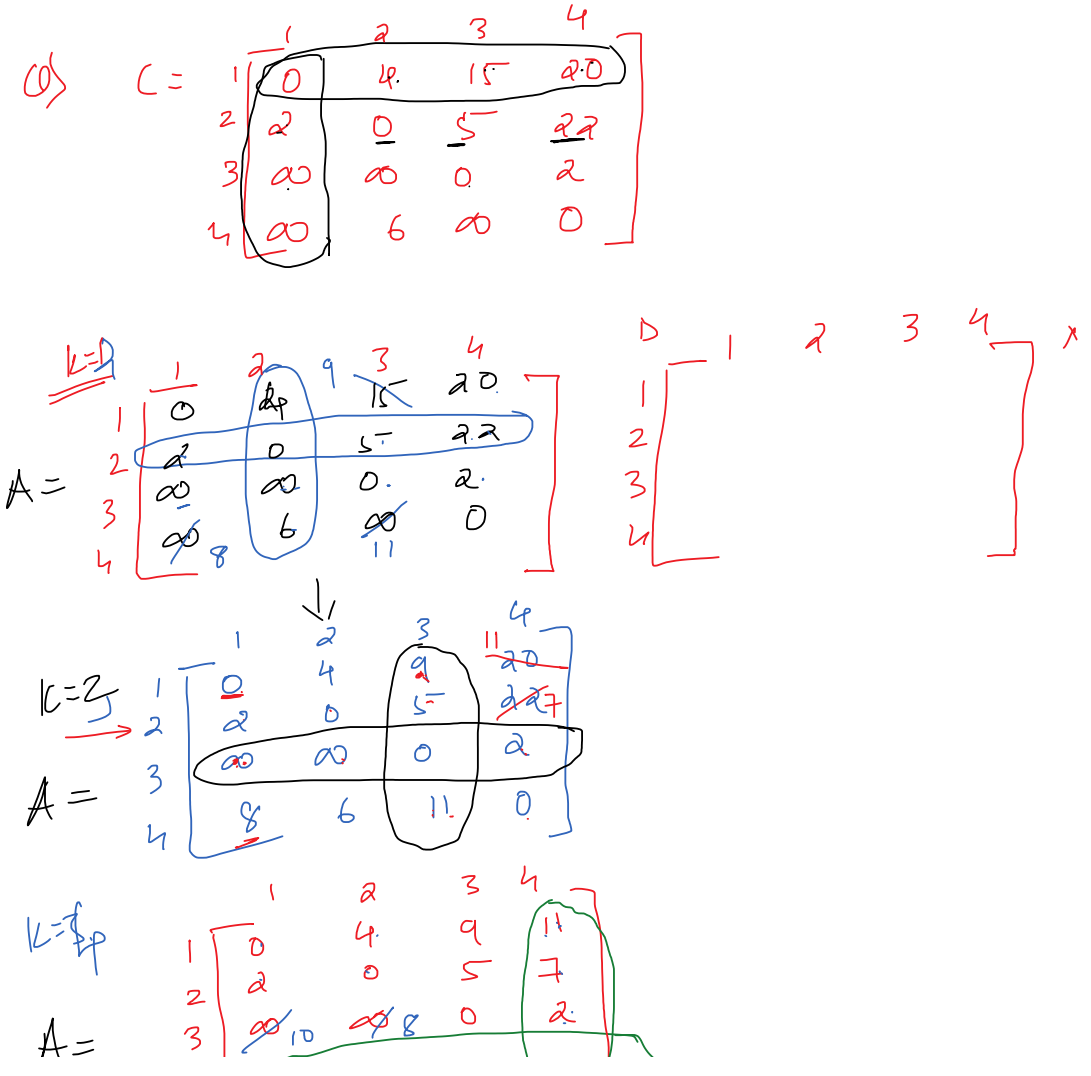
$k=2$ via 2



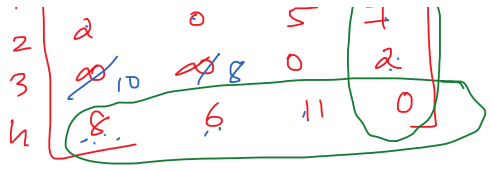
$$D = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} -1 & 1 & 2 \\ 2 & -1 & 2 \\ 3 & 1 & -1 \end{bmatrix} \end{array}$$

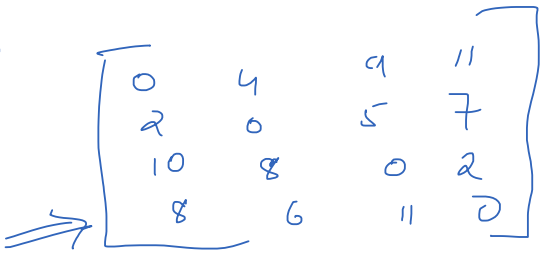
k 1 2 3


K=3
A=
Result
D
C (last)
D (via)
We can say C represents All pair Shortest path length


(1)
C=
K=1
A=
D
K=2
A=
K=3
A=

$A =$





Algorithm →

Floyd Warshall's Algo

function AllPairsShortestPath (C, A, n)

where n = no of vertices

C = Cost of Adjacency matrix

Ans — A = length matrix

```
1 start
2 for (i = 1 to n) do
3   for (j = 1 to n) do
4.    A(i,j) = c(i,j)
5=    for (k=1 to n) do
6     for (i=1 to n) do
7       for (j = 1 to n) do.
8.  A(i,j) = min( A(i,j) , A(i,k) + A(k,j) )
9   return.
```

Complexity = $O(n^3)$.

Bellman ford Algorithm · (SSSP) Single Source shortest Path (Dijkstras Algo).

Note: Dijkstras Algo is used to calculate Shortest path from source to dest^n

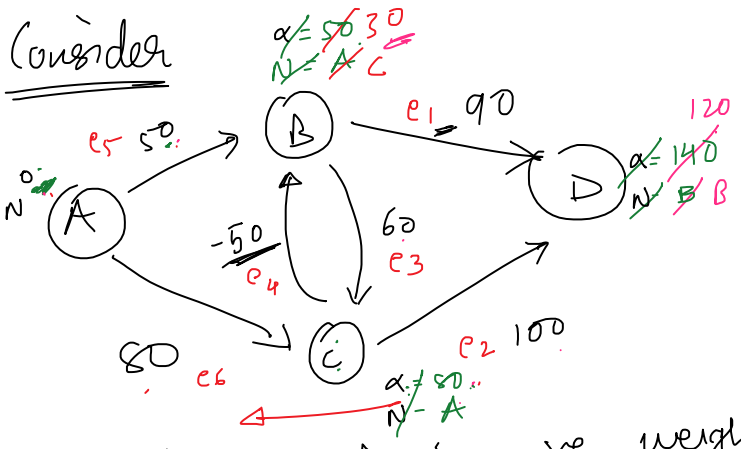
But if any edge of graph has -ve weight then Dijkstras Algo will fail.

* So if edge of a Graph is -ve then we will not use Dijkstras Algo, instead we will use Bellman ford Algorithm.

* Bellman ford can also be used when all the edges are +ve.

* If there is -ve weight cycle then Even Bellman ford Algo will fail

Consider



All pair shortest path from node A to all other node

Since edge C → B is -ve weight, we will use

Since edge C → B is -ve weight, we will use

Bellman ford Algo [Number the vertex from e1 → en from Sink Side]

	A	B	C	D
distance	~~∞~~ 0	∞	∞	∞
Start = A (source) path	NULL	NULL	NULL	NULL
i = 1 chk edge e1 → e6	0 NULL	50 A	80 A	∞ NULL
i = 2 chk from e1 → e6	0 NULL	30 C	80 A	140 B
i = 3. chk from e1 → e6	0 NULL	30 C	80 A	120 B
i = 4 check from e1 → e6	0 NULL	30 C (via A)	80 A	120 B

A → A => 0

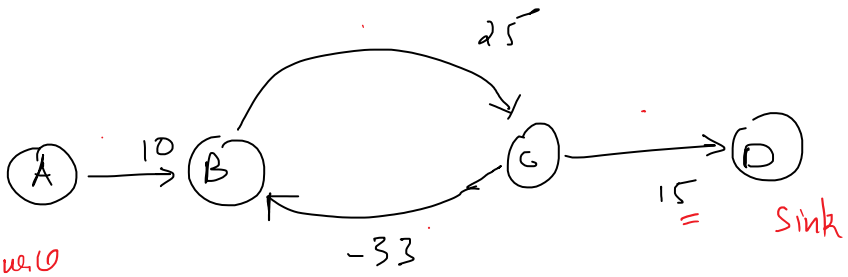
A → B => 30 via C.

A → C = 80

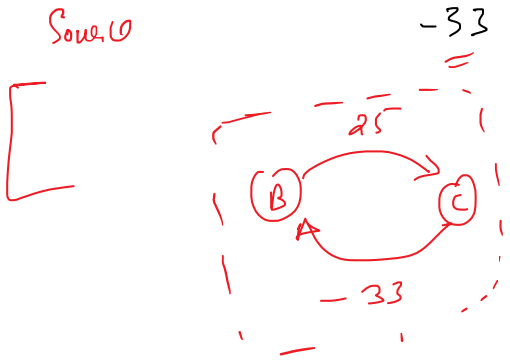
A → D → 120 via B

We solved it using Bellman ford

Consider An Example of -ve weight Cycle



Source



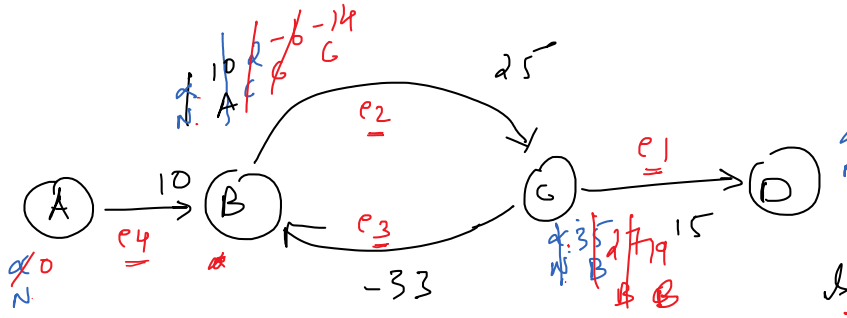
1st Round $B \to C \to B$ = cost = -8

2nd Round = -8 + 25 - 33 = -16

3rd Round = -16 + 25 - 33 = -24

Because there is -ve weight cycle, Every time going through cycle decreases the weight cost by 8.

In above graph there is -ve weight cycle



Source = A

4 vertex ⇒ 4 iterations

	A	B	C	D
Source A	~~∞~~ 0 NULL	∞ NULL	∞ NULL	∞ NULL
i=1	0 NULL	10 A	∞ NULL	∞ NULL
i=2	0 NULL	2 C	35 B	∞ NULL
i=3	0 NULL	-6 C	27 B	50 C
i=4	0 NULL	-14 C	19 B	42 C

From e_1 to e_4

} go from e_1-e_6

} go from e_1 e_6

} go from e_1-e_6

} go from $e_1 \to e_6$

Here every next iteration will give you cost

lesser than previous iteration

So the cost of shortest path from

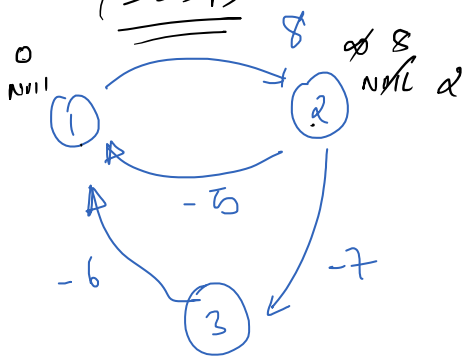
A → B ⇒ $-\infty$ (undefined)
A → C ⇒ $-\infty$ undefined
A → D ⇒ $-\infty$ Undefined.

If there is -ve weight cycle then Undefined is the Answer

Another Way to Solve (As per Gorman)

(SSSP)

Coincides ->



Source = 1

Pass 1 ⇒

Source

	1	2	3 ← vertex
d	0 ~~-5~~ ~~∞~~	~~∞~~ 8	~~∞~~ 1
	~~-13~~	~~-11~~	~~-12~~

(path)

Distn

Source
u = 1
~~d(u) = 0~~
~~π(u) = -1~~

V = 2
d(2) = ∞
d(u) + (u,v) = 0 + 8 < d(2) Yes
d(2) = 8 ✓
π(2) = 1 ✓

Current
u = 2
d(2) = 8 ✓

V = 1
d(1) = 0 ✓

V = 3
d(v) = ∞
d(u) + (u,v) = 8 + (-7) = 1 < ∞

u = 2
d(2) = 8 ✓
π(2) = 1 ✓

d(1) = 0
d(u) + (u,v) = 8 + (-5)
= 3 < d(v)
NO

d(v) = ∞
d(u) + (u,v) = 8 + (-7) = 1 < ∞
∴ d(3) = 1
π(3) = 2

u = 3
d(u) = 1
π(u) = 2

v = 1
d(v) = 0
d(u) + (u,v) = 1 + (-6) = -5 < d(v) Yes
d(1) = -5 ✓
π(1) = 3 ✓

	1	2	3
d	-5 -10	8, 3	4 -4
π	3 3	1 1	2

Pass 2.

u = 1
d(u) = -5
π(1) = 3

v = 2
d(v) = 8
d(u) + (u,v) = -5 + 8 = 3 < 8
d(v) = d(2) = 3
π(v) = π(2) = 1

u = 2
d(u) = 3
π(u) = 1

v = 1
d(1) = -5
d(u) + (u,v) = 3 + (-5) = -2 ≠ -5
NO
change

v = 3
d(v) = d(3) = 1
d(u) + (u,v) = 3 + (-7)
= -4 < 1
d(3) = -4
π(3) = 2

u = 3
d(3) = -4
π(3) = 2

v = 1
d(1) = -5
d(u) + (u,v) = -4 + (-6) = -10 < -5 Yes ✓
d(1) = -10 ✓

$\pi(3) = 2$ //

$d(1) = -10$ ✓

$\pi(1) = 3$ ✓

	1	2	3
d	-10	~~-2~~ -2	~~-6~~ -9
π	3	1	2

Pass 3 →

$u = 1$

$d(u) = -10$

$\pi(u) = 3$

$v = 2$

$d(2) = 3$

$d(u) + (u,v) = 8 - 10 = -2 < 3$ Yes

$-10 + 8 = -2$ ✓

$d(2) = -2$ ✓

$\pi(2) = 1$

$u = 2$

$d(2) = -2$

$\pi(2) = 1$ ✓

$v = 1$ //

$d(1) = -10$

$d(u) + (u,v)$

$= -2 - 5 = -7 \not< -10$

No change //

$v = 3$

$d(3) = 2$

$d(u) + (u,v)$

$= -2 - 7 = -9$ <- Yes

$d(3) = -9$ ✓

$\pi(3) = 2$ ✓

$u = 3$.

$d(3) = -9$

$\pi(3) = 2$

$v = 1$

$d(1) = -10$

$d(u) + (u,v) = -6 - 9 = -15 < -10$ Yes

$d(1) = -15$ ✓

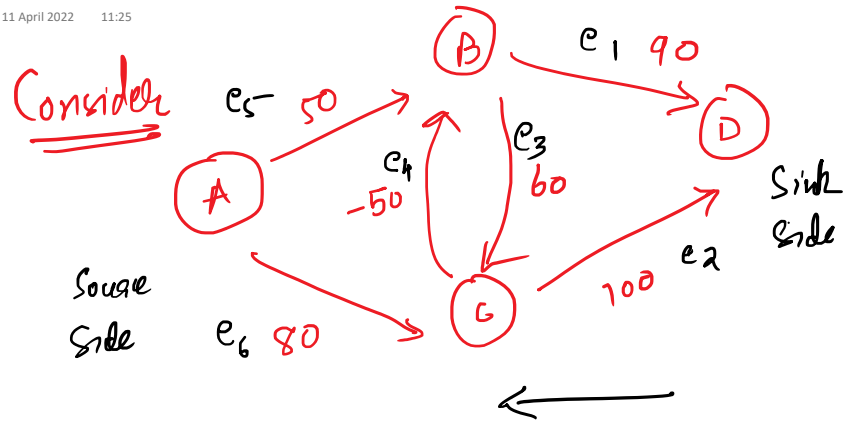
$\pi(1) = 3$. ✓

	1	2	3
d	-15	-2	-9
π	3	1	2

The d in 1, 2, 3 vertex goes on decreasing

On every iteration

there is -ve weight cycle in graph.

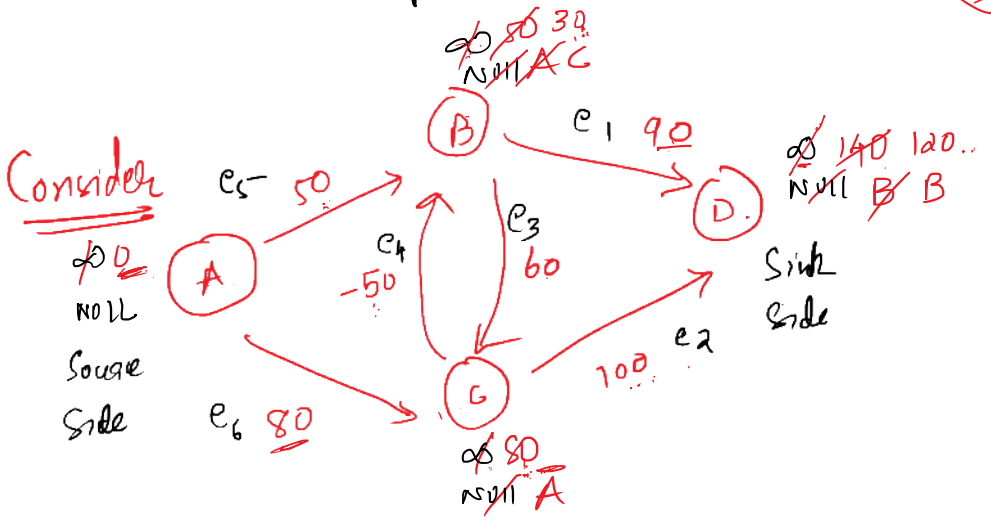


To find Single Source Shortest Path from Source A

Since cost of edge C→B is -ve i'e -50

we will not use Dijkstras Algo, instead will use Bellman ford Algorithm.

Note : Number the edges from e1 to en from Sink Side



		A	B	C	D
Source A		~~∞~~ 0 NULL	∞ NULL	∞ NULL	∞ NULL.
Jo from e1 to e6	i=1	0 NULL	50 A	~~80~~ A	∞ NULL
go from	i=2	0 NULL	30 C	80 A	140. B

go from e_1 to e_6	$i=2$	0 NULL	30 C	80 A	140 B	
go from e_1 to e_6	$i=3$	0 NULL	30 C	80 A	120 B	
go from e_1 to e_6	$i=4$	0 NULL	30 C	80 A	120 B	Nothing changed so stop.

from A to B = 30 via C
A to C = 80 via A
A to D = 120 via B

Bellman ford →

* If a graph with n vertices and $n \geq 6$ then we need to check for 2 to 3 passes and if the value of array d remains same for any 2 consecutive pass then we will conclude that there is no negative weight cycle in the graph. * Else we ideally need to go for n passes

Algo for Bellman ford.

```
fn BellmanFord (G, w, s)
{   G => graph.
    w = weight matrix
    s = starting vertex.
    Integer d(1:n) and π(1:n).
    d → cost,  π → path
1.  Start
2.  for each vertex u ∈ V(G) do
    {
3.      d(u) = ∞
4.      π(u) = null (-1)   OR
    }
5   d(s) = 0        (s → starting vertex)
6   for (i = 1 to n-1) do
    {
7.      for each edge (u,v) ∈ E(G) do
8           if (d(u) + w(u,v) < d(v)) then
9.              d d(v) = d(u) + w(u,v)
    }
```

u=1 , v=2

```
        d d(v) = d(u) + w(u,v)
        π(v) = u
    }
}
for each edge (u,v) ∈ E(G) do
    if ( d(u) + w(u,v) < d(v) )
        print "Negative weight cycle"
        return
print "NO -ve weight cycle"
return
}
```

0/1 Knapsack Using Dynamic Programming

Fractional Knapsack ⇒ fraction of weight of object is allowed.

- ✓ If object ⇒ weight = 5 kg
- ✓ If Remaining capacity = 2 kg

We are allowed to take 2 kg out of 5 kg

0/1 Knapsack ⇒ You are supposed to consider object with its whole weight and fraction is not allowed

Consider ⇒ capacity = 5

No of object = 4.

w = { 1, 3, 2, 2 }

p = { 4, 11, 6, 14 }.

$C[i][j] = \max(C[i-1][j], 11 + C[1][2-3])$

Either fill it with object 1

object capacity

cap

object 2

$C[3][3] = (C[2][3], 6 + C[2][3-2])$

object capacity 1,2,3

11 , 6 + 4 = 10

Using 0/1 Dynamic Knapsack

j ⇒ capacity

object ↓	0	1	2	3	4	5
1	0	4	4	4	4	4
2	0	4	4	11	15	15
3	0	4	6	11	15	17
4	0	4	14	18	20	25

profit

Now → from above matrix.

Max profit = 25

Here chk for object 4 → c[4][5] ≠ c[3][5]

Hai Ya Nahi

c[4][5] ↑ profit when object 4 for capacity 5

c[3][5] ↑ profit when obj 3 for capacity 5.

Object 4 is added in solution

weight of object 4 is 2

Remaining capacity ⇒ 5 − 2 = 3

To check for object 3

c[3][3] = c[2][3]

= 11 = 11

} means obj 3 is not part of the solution.

Now chk for obj 2

c[2][3] ≠ c[1][3]

= 11 = 4

} Object 2 is considered in solution

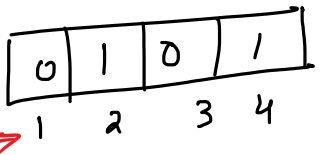
Remaining capacity = 3 − 3 = 0

The object to satisfy the capacity object 2 ↓ 3, object 4 ↓ 4

Profit = 11 + 14 = 25 ✓

Soln *

Soln ✱ away



object no

⇒ 1 2 3 4 : 0 1 0 1 ✓ Ans

$C[i][j]$ = gives profit when object i satisfies capacity j

In matrix G

i = object number

j = capacity

$$C[i][j] = \max\left(C[i-1][j] \;,\; P[i] + C[i-1][j - w[i]] \right)$$

$C[i-1][j]$: Pervious obj satisfying j capacity — i^{th} object not used

$P[i]$: object i is considered so write profit

$C[i-1]$: pervious obj

$j - w[i]$: pervious capacity [removing obj i] — i^{th} object used

weight of i^{th} obj

if $j - w[i] \geq 0$

& $C[i][j] = C[i-1][j]$ if $j - w[i] < 0$

Consider ⇒

No of object = 4

Capacity = 4

w = { 2, 1, 3, 2 }

P = { 8, 6, 15, 14 }

0/1 Dynamic Knapsack.

0/1 Dynamic Programming

	0	1	2	3	4
1	0	0	8	8	8
2	0	6	8	14	14
3	0	6	8	15	21
4	0	6	14	20	22

c[4][4] ≠ c[3][4]

obj'd 4 is added

Remain = 4 - 2 = 2

c[3][2] == c[2][2]

obj 3 is not considered

c[2][2] == c[1][2]

obj 2 is not considered

obj 1 is considered

Remain = 2 - 2 = 0

1	2	3	4
1	0	0	1

obj 1 & obj 4 is considered

Note

c[3][3] =

↑ I have objects 1, 2 & 3 ↑ capacity

c[2][3], 6 + c[2][3-2] (= 1)

↑ using 1, 2

obj 3 not used.

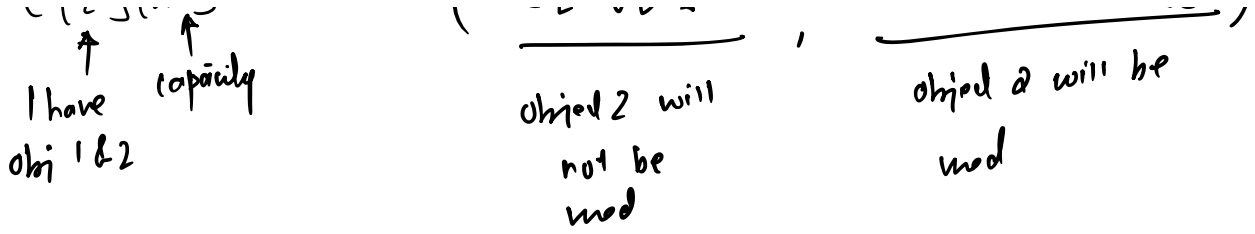
obj 3 used.

c[2][2] = (c[1][2] , 11 + c[ ][2-3] (-1 < 0))

↑ ↑ capacity

obj 2 will

obj 2 will be

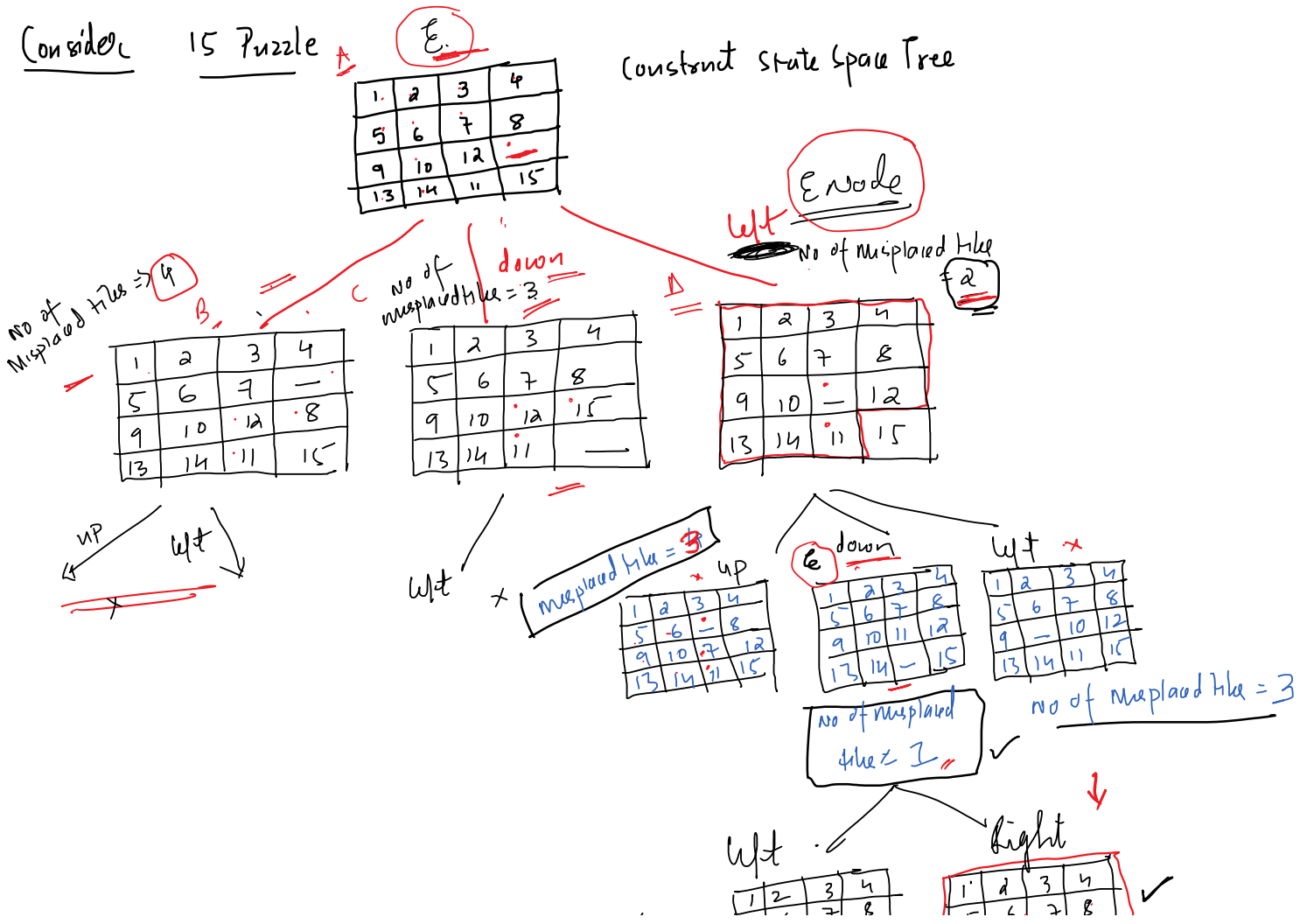

I have
obj 1 & 2
capacity
object 2 will
not be
mod
object 2 will be
mod

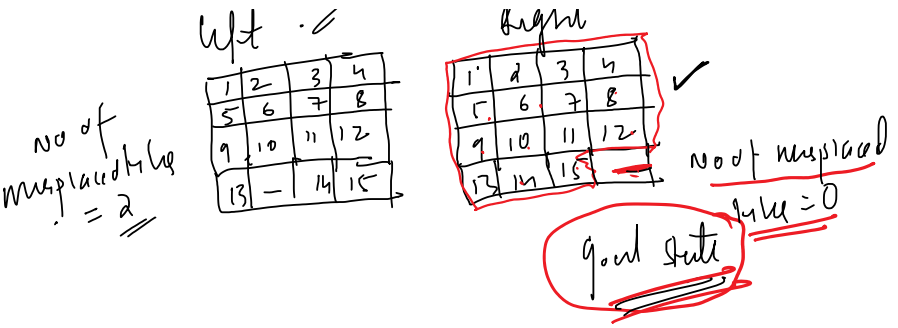
Branch and Bound.

It is a technique of exploring a directed graph or State Space Tree

* In this we calculate bound (No of Misplaced Tiles) on each node which represents the cost of reaching to goal solution (final) node from that node.
* If the solution represented by bound, is worst than best solution found so far then that node is killed and corresponding branch is closed
* The bound can also be used to select most promising node out of available

Consider 15 Puzzle



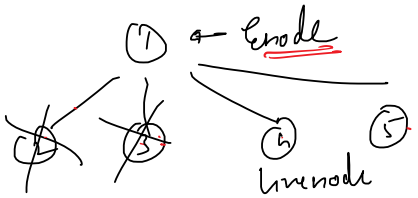


Approach

E-Node ⇒ It is a node that is currently getting Expanded.

Live Node ⇒ It's a node which is created but not Expanded yet. In Branch and Bound at each E node we find all the live nodes that can be generated using single step

Those which are infeasible are killed and other are added to the list of live node



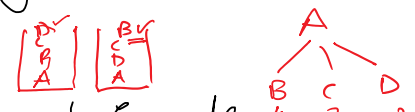
The following rules are followed in B & B ⇒

(1) All the children of current E node are generated before selecting the next E-node

(2) Each node may become an E-node exactly once.

(3) Depending on policy of selecting next E-node B & B can be categorised as

(A) LIFO BB/LIFO Search ⇒ Here next E node is selected (Stack) using DFS.



(B) FIFO BB / FIFO Search ⇒ Here next E node is selected using BFS.

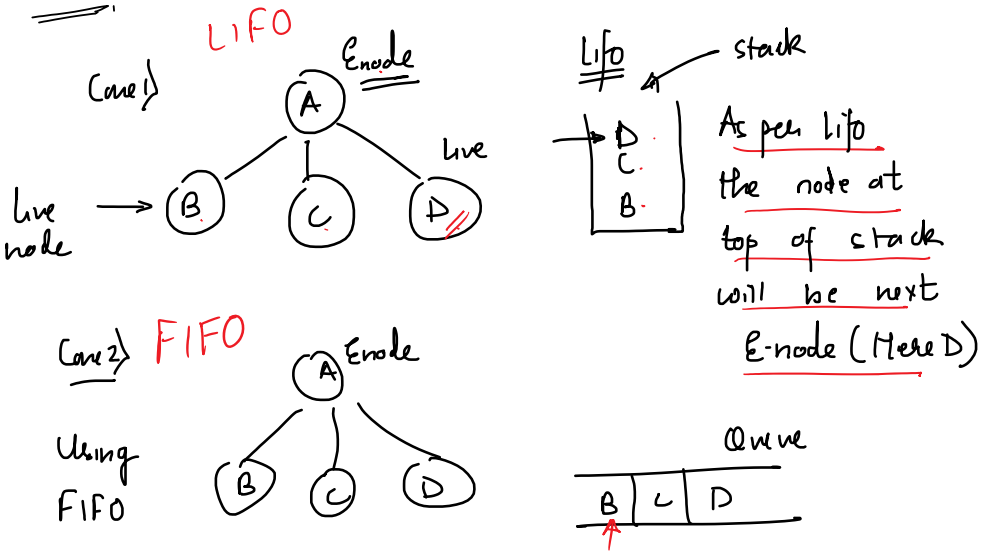
Queue

(C) LC BB / LC Search [LC = Least Cost] ✓

We use this

In this next E-node is selected from the Live node having least cost.

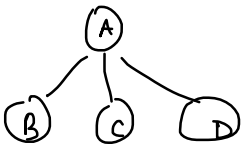
* LC Search uses Estimated Cost [ĉ(.)]



As per operation on queue data structure the node B will be E-node

[But Both the above are not Efficient]

We Use Least Cost



Next E-node = min(B, C, D) cost

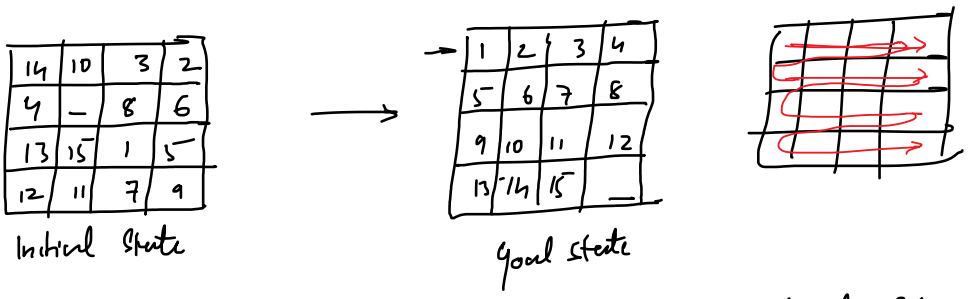
⇒ It is Intelligent methodology and is efficient

Puzzle Problem

In this we have a square block frame of 4×4

dimension with 15 numbered tiles and an empty slot (ES)

The problem is to reach to goal state from Initial State



Initial State

Goal state

* A legal move can be represented as movement of ES
* Here there are maximum 4 legal moves possible
  UP, DOWN, LEFT and RIGHT
* Hence we want to find set of legal moves to transform the initial state to goal state.
* A goal is reachable from initial state if and only if the initial state is reachable from goal state
* The Reachability can be as follows:

Checking Reachability

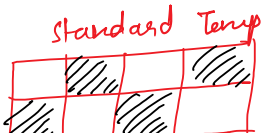
The initial state is reachable from goal state if

$$\sum_{i=1}^{15} less(i) + X \text{ is Even}$$

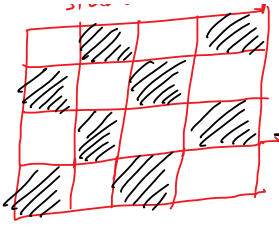
Here $less(i)$ = No of tiles having value $< i$ but they are ahead of it.

X can be computed by comparing initial state with following board arrangement

Standard Template



→ Even col shaded
→ odd " "



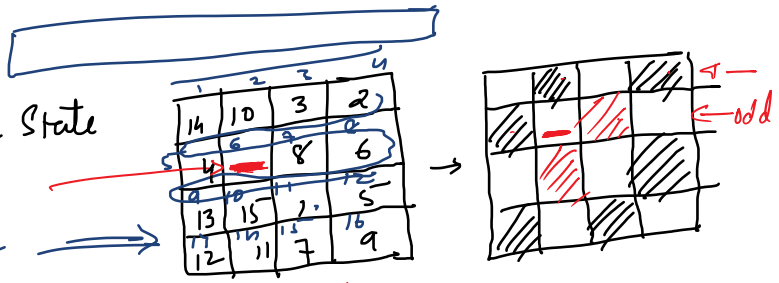
✓

Empty slot

$x=1$ if ES in initial state matches with shaded slot

$x=0$ otherwise

Consider the Initial State



See here

Initial state

Empty slot

Here the ES in Initial state does not matches with shaded slot So $\boxed{x=0}$ ✓

Now To Calculate $\Rightarrow \sum_{i=1}^{15} \text{leas}(i)$

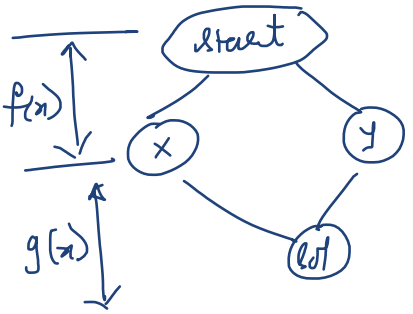
i	leas(i)
1	0
2	1
3	2
4	1
5	0
6	2
7	0
8	4
9	0
10	9
11	2
12	3
13	6
14	13
15	6

49

$\sum_{i=1}^{15} less(i) + X = 49 + 0 = 49$ Not Even

The Goal state Here is not reachable

Compute $c\hat{}()$



$$c\hat{}(x) = f(x) + g(x).$$

In 15 puzzle problem

$f(x)$ = depth of node X.

$g(x)$ = no of non blank tiles not in their goal state

Travelling Salesman Problem Using Branch and Bound.

Approach :→ A better estimated cost function using "Reduced Cost Matrix"

[* A row of matrix is said to be reduced if it contains atleast one zero and remaining elements are Non Zero]

Ex → [10 2 7 ∞]

↓ To Reduce the row subs 2

[8 0 5 ∞] → reduced row.

Similarly col can also be reduced.

* A matrix is said to be reduced if all rows and all columns are reduced.

To find $\hat{C}(n)$ ⇒ Cost to reach destination from Current node n.

Steps →

Step1> set all elements of i^{th} row and j^{th} col to ∞

Step2> set A[j,1] = ∞

Step3> Reduce all rows and col of Resultant matrix. except the row & col containing ∞

step 4) Compute the total Reduction value

$$\hat{c}(s) = \hat{c}(R) + A(i,j) + r$$

- $\hat{c}(R)$: s will be child of R from R → S
- $A(i,j)$: cost of edge i,j
- r : Total reduction value. (Row + Col).

S = current node edge from i to j

R ⇒ from R to current node(s).

19 April 2022 11:35

Dec-14 Consider

	1	2	3	4
1	∞	10	15	20
2	5	∞	9	10
3	6	13	∞	12
4	8	8	9	∞

Solve TSP using Branch and Bound.

Solution → Given Cost Matrix =

$$\begin{bmatrix} \infty & 10 & 15 & 20 \\ 5 & \infty & 9 & 10 \\ 6 & 13 & \infty & 12 \\ 8 & 8 & 9 & \infty \end{bmatrix} \quad (1)$$

To find $\hat{c}(1)$ = cost to reach dest^n node from node 1. } Since the tour starts from 1 so here there is no R

→ Go for Direct Reduction.

for Row Reduction →

10 →	∞	10	15	20
5 →	5	∞	9	10
6 →	6	13	∞	12
8 →	8	8	9	∞

29

Total Row Reduction =>

	∞	0	5	10
	0	∞	4	5
	0	7	∞	6
	0	0	1	∞
	0	0	1	5

⇒ 6 Total col Red^n

Now col Reduction →

$\hat{c}(1)$ =

	1	2	3	4
1	∞	0	4	5
2	0	∞	3	0
3	0	7	∞	1
4	0	0	0	∞

Total Red => 29 + 6 = 35.

Step 2 → find $C^\wedge(2)$ → Cost of reaching dest^n node from node 2.

Here $i = 1$, $j = 2$

In $C^\wedge(1)$ → set i^{th} row to ∞

set j^{th} col to ∞

Set $A(j,1) \to \infty$ ∴ $A(2,1) = \infty$

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

⇒ 0 col^n Red^n.

Row Red^n → Total Row Red^n ⇒ 0

col Red^n ⇒ Total col Red^n ⇒ 0.

$k = 0$

$A(i,j)$ = Cost of edge $1 \to 2$.

$A(1,2)$ = In $C^\wedge(1)$ cost $1 \to 2 = 0$

$C^\wedge(2) = C^\wedge(1) + A(i,j) + \gamma$ → Total Red^n of $C^\wedge(2)$

$= 35 + 0 + 0$

(35 ↑ Total red^n of $C^\wedge(1)$)

$= 35 + 0 + 0 = 35$

$C^\wedge(3)$ → Cost to reach final node from node 3

$i = 1$, $j = 3$

In $C^{\wedge}(1)$ → make i^{th} (1) row $\Rightarrow \infty$
j^{th} (3) col $\Rightarrow \infty$

make. $(j,1) = \infty$
$(3,1) \Rightarrow \infty$

$$\therefore C^{\wedge}(3) = \begin{array}{c} 0 \\ 0 \\ 1 \\ 0 \\ \hline 1 \end{array} \begin{bmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & 0 \\ \infty & 7 & \infty & 1 \\ 0 & 0 & \infty & \infty \end{bmatrix}$$

Row Red^{n} -> Total Row Red^{n} = 1

$$\begin{bmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & 0 \\ \infty & 6 & \infty & 0 \\ 0 & 0 & \infty & \infty \\ \hline 0 & 0 & 0 & 0 \end{bmatrix}$$

Col Red^{n} = 0

Total Red^{n} = r = 1+0 = 1.

$A(i,j) = \text{In } C^{\wedge}(1) = (1,3) = 4$

$\therefore \hat{C}(3) = \hat{C}(1) + A(i,j) + r$
$35 + 4 + 1 = 40$

$C^{\wedge}(4)$ = Cost of reaching destination from node 4

i	j
1	4

In $C^{\wedge}(1)$ make i^{th} row => 1st row to ∞
j^{th} col => 4th col to ∞

also $(j,1) = (4,1)$ to ∞

also $(i,j) = (4,1)$ to ∞

0	∞	∞	∞	∞
0	0	∞	3	∞
0	0	7	∞	∞
0	∞	0	0	∞
	0	0	0	0

Row Red^n $\Rightarrow 0$

Col Red $\Rightarrow 0$ Total Red^n = $R = 0$.

In $\hat{c}(1)$ $(i,j) = (1,4) \Rightarrow 5$

$\therefore \hat{c}(4) = \hat{c}(1) + A(i,j) + R$

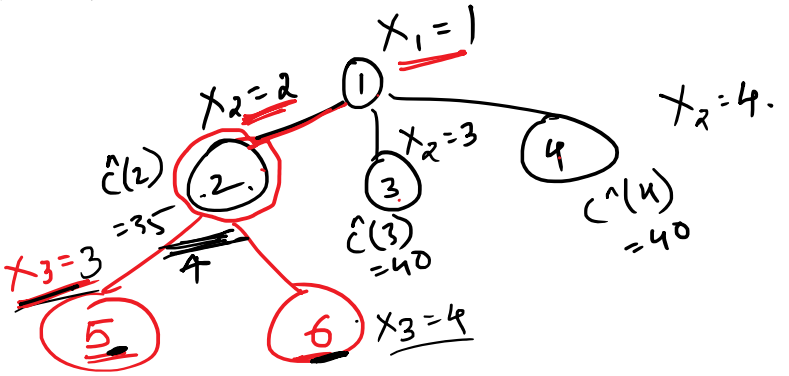
$= 35 + 5 + 0 = 40$ //.

Now $\hat{c}(2)$ = cost to dest^n from 2 $\rightarrow$ visiting 2 from 1 $\Rightarrow$ 35

$\hat{c}(3)$ = " " " " 3 $\rightarrow$ " 3 " " $\Rightarrow$ 40

$\hat{c}(4)$ = " " " " 4 $\rightarrow$ " 4 " " $\Rightarrow$ 40.

least cost is $\hat{c}(2)$ ie In SSST we will visit node 2.



Step 3 Now we visited from vertex 1 to vertex 2

Now at vertex 2 ie $\hat{c}(2)$ we have options to

visit vertex 3 $\Rightarrow \hat{c}(5)$

or visit vertex 4 $\Rightarrow \hat{c}(6)$

$\hat{c}(5)$ = from node 2 to 3

In $\hat{c}(2)$ ⇒ make 1^{st} row ⇒ ∞ (2)

make j^{th} col ⇒ ∞ (3)

$(j,1) = (3,1) = \infty$

$$\begin{matrix} 0 \\ 0 \\ 1 \\ 0 \end{matrix} \begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & 1 \\ 0 & \infty & \infty & \infty \end{bmatrix}$$

⇒ 1

Total Row Red^n = 1

$$\begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & 0 \\ 0 & \infty & \infty & \infty \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Col^n Red^n = 0

Total Red^n = 1+0 = 1 $\quad r_2 = 1$

In $\hat{c}(2)$ ⇒ (2,3) ⇒ 3

$\hat{c}(5) = \hat{c}(2) + (2,3) + r$

$= 35 + 3 + 1 = 39$.

Now $\hat{c}(6)$ = In Tree from node 2 to 6

⇒ 2 → 4

In $\hat{c}(2)$ make 2^{nd} row = ∞

4^{th} col = ∞

make (4,1) = ∞

$$\begin{matrix} & 1 & 2 & 3 & 4 \\ 1 & 0 & \infty & \infty & \infty & \infty \\ 2 & 0 & \infty & \infty & \infty & \infty \end{matrix}$$

$\hat{c}(6) \Rightarrow$

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

Row" $\Rightarrow 0$

col Red" $\Rightarrow 0$

$r_2 = 0+0 = 0$

In $\hat{c}(2) = (2,4) = 0$

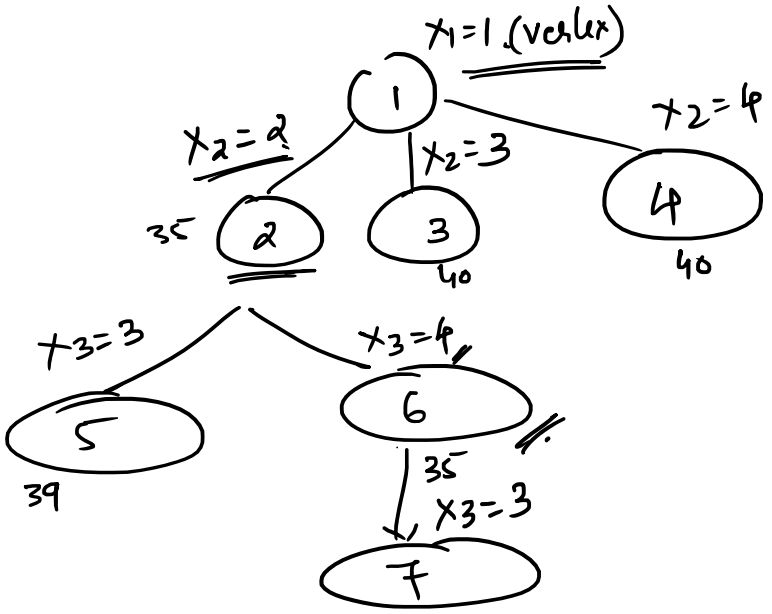
$\hat{c}(6) = \hat{c}(2) + (2,4) + 0 = 35$

$35 \quad 0 \quad 0$

From (vertex 2) node 2 in Tree — vertex 3 — node 5 of tree $\Rightarrow 39$

vertex 4 — node 6 of tree $= 35$

We will visit vertex 4 from vertex 2.



Now at vertex 4 $\Rightarrow$ only option is vertex 3

From $\hat{c}(6)$

0	1
4	3

$\hat{c}(7) =$

In $\hat{c}(6)$ $\quad 4^{th}$ row $= \infty$
3^{rd} col $= \infty$

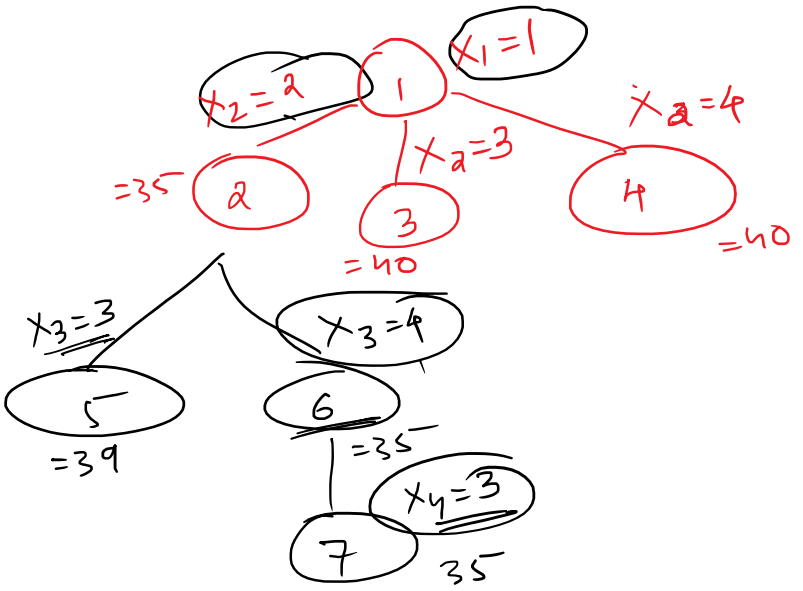
$(3,1) \longrightarrow \infty$

$$\begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \end{bmatrix}$$

Row Red $= 0$
col Red $= 0$

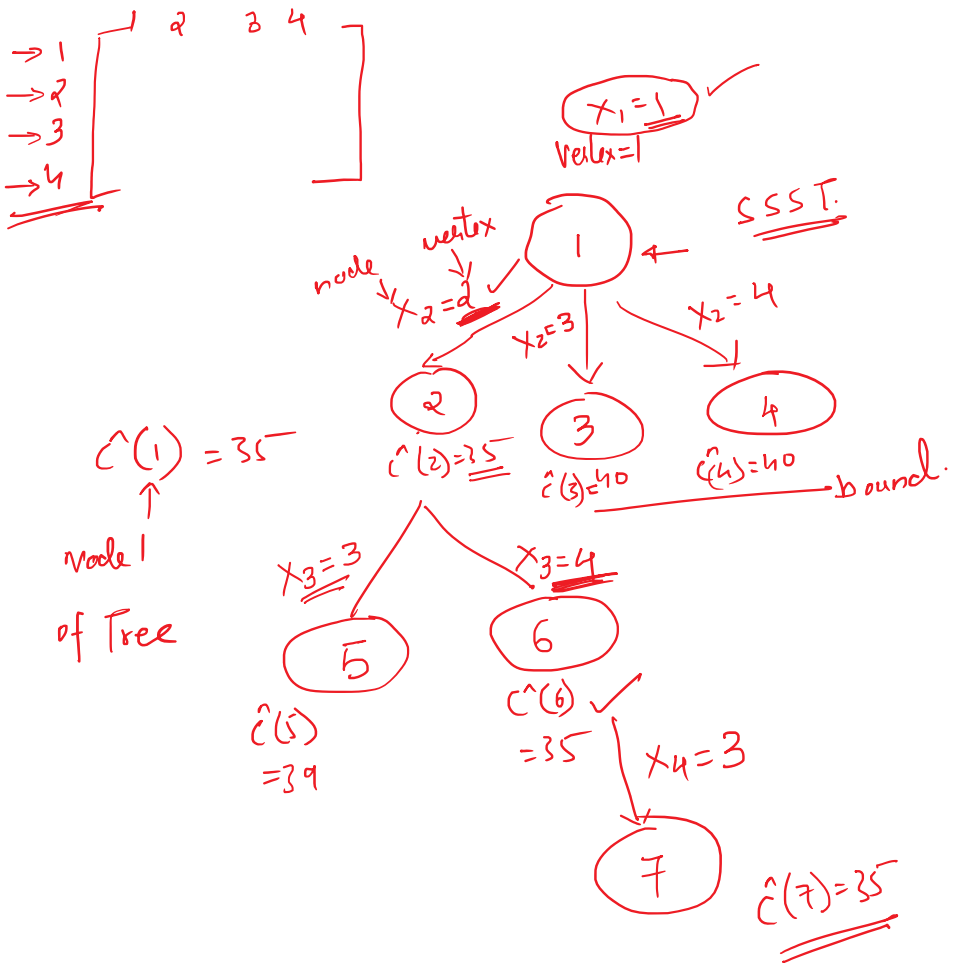
$d_4 = 0 + 0 = 0$

$\hat{c}(7) = \hat{c}(6) + (4,3) + d_4$ (In $\hat{c}(6)$)
$= 35 + 0 + 0 = 35$



Path $= 1 \to 2 \to 4 \to 3 \to 1$
$10 + 10 + 9 + 6 = 35$

$\hat{c}(7) = 35$.



Trick $\hat{c}(1)$ ⟶ only row & col^n Reduction.

Refer $\hat{c}(1)$

- $\hat{c}(2)$ = from i to j
- $\hat{c}(3)$ from i to j
- $\hat{c}(4)$ from i to j

i^{th} row $= \infty$

j^{th} col $= \infty$

$[j, 1] = \infty$

$\hat{c}(2) = \hat{c}(1) + (i, j) + r$

$\hat{c}(3) = \hat{c}(1) + (i, j) + r$

$\hat{c}(4) = \hat{c}(1) + (i, j) + r$.

Refer (i, j) in $\hat{c}(1)$

Here we selected $\hat{c}(2)$ as it was least cost

Now 1-2 done we need to decide from 2 ⟶ 3, 4

Now 1-2 done we need to decide from 4

Refer $\hat{c}(2)$

$\hat{c}(5)$ = from i to d : 2 to 3

$\hat{c}(6)$ from i to d : 2 to 4

i^{th} row, j^{th} col $\} \infty$

$[j,1] = \infty$

$\hat{c}(5) = \hat{c}(2) + (i,j) + r$

$\hat{c}(6) = \hat{c}(2) + (i,j) + r$

Refer (i,j) in $\hat{c}(2)$.

Here we select $\hat{c}(6) = 35$ as least cost

So uptill now $1 \to 2 \to 4$, Now we are at 4^{th} vertex

from 4 will visit 3 only.

Refer $\hat{c}(6)$ $\hat{c}(7)$ = from 4 to 3 : i to d

i^{th} row, j^{th} col $= \infty$

$(j,1) = \infty$

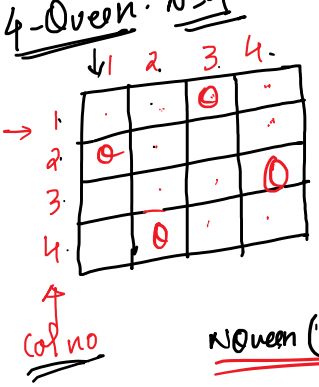
Backtracking Approach :-

NQueen Problem →

Given. { N × N chessboard
N Queen

[None of the queens are attacking position]

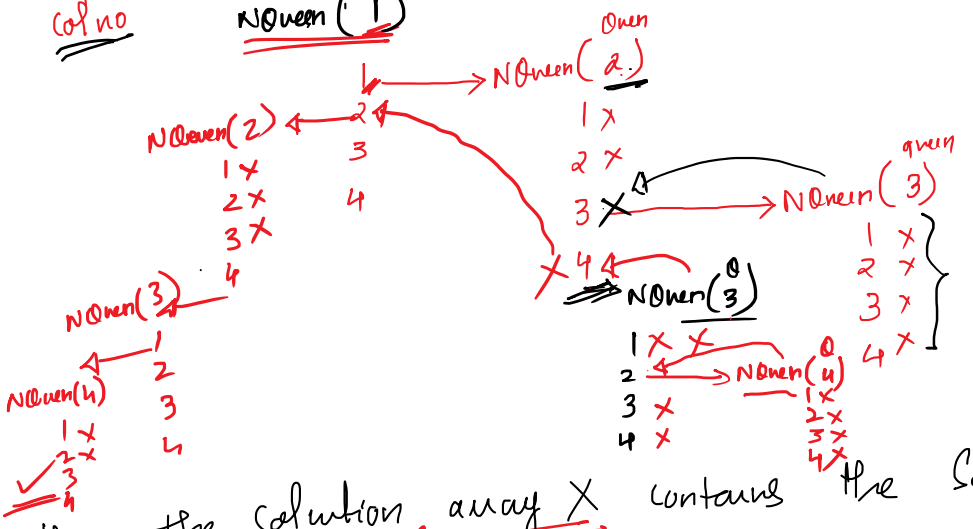
4-Queen. N=4



← Queen

X | 2 | 4 | 1 | 3 |
1 2 3 4 ← queen nos

Queen.



Here the solution array X contains the solution

Here X[0] gives column position of queen 0

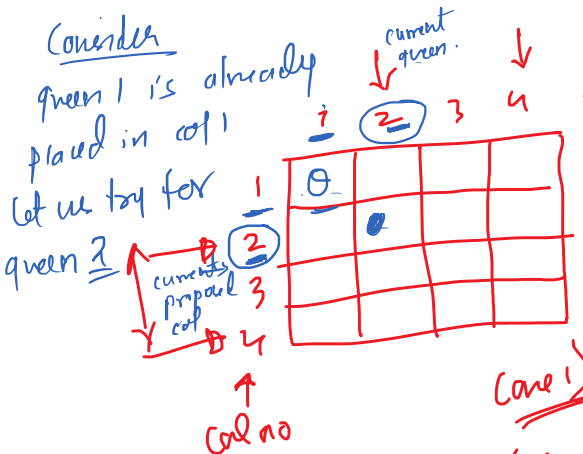
X = | 2 | 4 | 1 | 3 | ← Solution array X
1 2 3 4

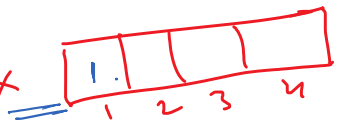
① Every value in array X must be unique
[Every queen must be placed at diff col^n]

② Every value must be

between 1 to 4 (inclusive)

Consider queen 1 is already placed in col 1 let us try for queen 2





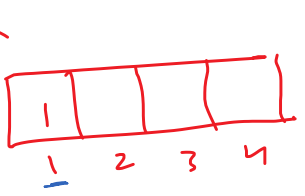
queen no 2, col no 1 → place(2, 1) } Some col Hai ya Nahi

Case 1) chk for previous queen ⇒ j = 1 ✓ (j = k-1, previous queen)

x[j] i.e. x[1] = 1 ⇒ poorane queen ka col and i = 1 ⇐ current queen ka proposed col same hai

Nahi chalega.

Case 2) chk diagonal



place(2, 2) (k, i) — queen, col

Previous queen = j = 1 (k-1)

Here x[j] = 1 i = 2 not Same ✓

col of previous queen | x[j] - i | == | k - j |

| 1 - 2 | == | 2 - 1 |

1 == 1 Same hai } diag attack hai

Current & poorane queen no ka diff = poorane and Current queen ke col ka diff (diag)

Case 3 → place(k = 2, i = 3)

① chk for same col → j = k-1

x[j] = x[i] = 1

pervane queen ka Alloted col

i = 3

current queen ka proposed col

= 1

Same Nahi Hai

② chk diag

= 2

| x[j] − i |

| 1 − 3 |

pervane queen ka col.

Naye queen ka proposed col

| k − j |

| 2 − 1 |

current queen

pervane queen

Above not same

So k^{th} queen can be placed at i^{th} col.

BACKTRACKING

★ Backtracking :

- It is a method of solving a problem (Eg. N Queen)
- The principle idea is to construct solutions by taking one component at a time. The component is added in the solution if all the constraints in the problem definition are satisfied.
- If constraints are not satisfied then the latest component is dropped from the solution.

i.e. while finding the solution to the problem, we find some partial solution. step 1, step 2, ... step k

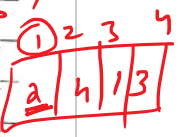
- At step k+1, the program discovers that it cannot go further with the solution due to some mistake in the previous step in such situation the program can be made to backtrack and repair the previous step solution.

★ N-QUEEN PROBLEM :→

- n x n chess board and given n queen.
  We want to place all the n queen on the board in non-attacking position.

Note: Queen can attack horizontally, vertically & diagonally.

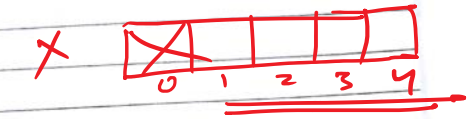
- Solution can be found using 1-D array of 'n' elements.
- k^{th} element of array x will give column of k^{th} queen
- The constraints on array x are every value stored in ary. x stored should be unique.
- Every value stored in array x should be between 1 to n.



Algorithm : for N Queen.

1. Let n be number of queens ✓
2. Let x[n+1] be array that stores column of queen.

n = 4



Algo for function read()

i. Accept no. of queens. i.e. n ✓
ii. create array x of n+1 elements.

Algo for function place (k, i)

It indicates whether kth queen can be placed in ith column.

```
int place (int k, int i)        // k: queen NO, i: col
{
  int j;
  for (j=1; j<=k-1; j++)        // j refers to previous queens.
  {
    if ((x[j]==i) || (abs(x[j]-i)==(k-j)))
      return 0;
  }
  return 1;
}
```

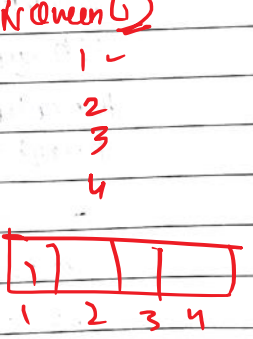
queen ke col ka diff

queen ke diff

Algo for function nqueen (int k)

```
void nqueen (int k)             // k: queen NO
{
  for (i=1; i<=n; i++)          // for all n col
  {
    if (place (k,i))            // Yes
    {
      x[k]=i;                   // place karo
      if (k==n)                 // is it last queen
      {
        for (j=1; j<=n; j++)    // To print solution.
          print x[j];           // print x array.
      }
    }
  }
}
```

nQueen()



```
else
    nqueen(k+1);   → agar queen no n nahi hai (last queen nahi hai)
  } // end of if   ← call for next queen.
} // for ends
} // end of nqueen.
```

NOTE:

- j refers to previous queens. (In place function.)
- x[j] : column of previous queen j. (allotted to)
- i : purposed column of current queen k. (i = current queen; i = proposed col^n of current queen.)
- |x[j]-i| : difference in column of previous & current queen.
- (k-j) : difference in queen number.

```
if((x[j] == i) || (|x[j]-i| == (k-j)))
```

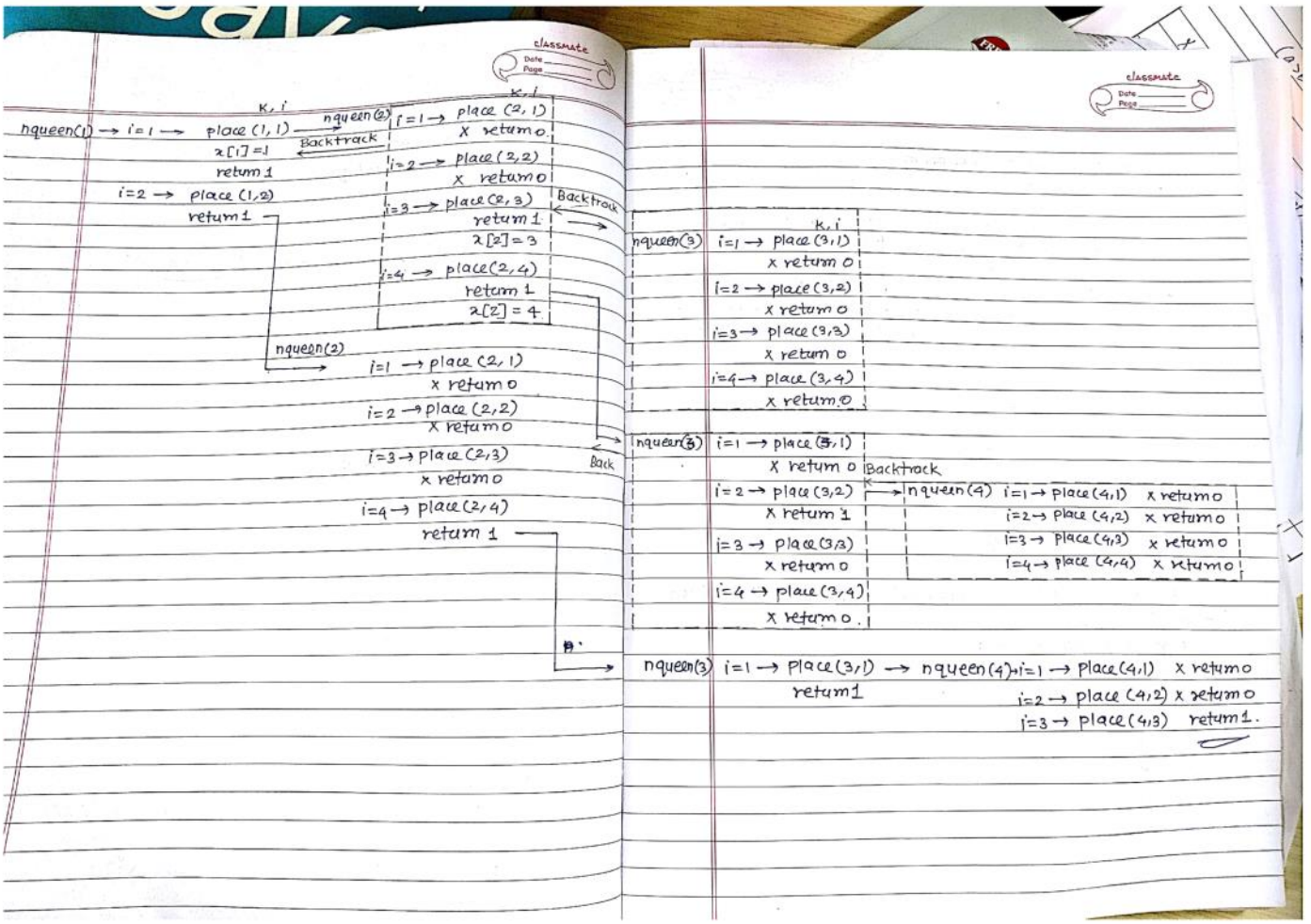
- x[j]: previous jth queen ka column number.
- i: current kth queen ka column number (proposed)
- |x[j]-i|: Previous jth queen ka column and current kth queen ka col. ka difference.
- (k-j): kth queen ka and previous jth queen ka difference.

Agar column no. same hai to Nahinn chalega !!

Agar same hai to diagonal problem Hai, Nahinn ! chalega

PROGRAM:

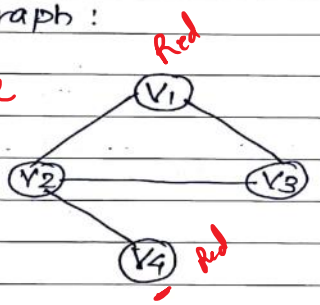
```
# include <stdio.h>
int n, x[16]={0};
int place (int k, int i)
{
  int j;
  for (j=1; j<=k-1; j++)
    if ((x[j]==i) || (abs(x[j]-i)==(k-j)))
      return 0;
  return 1;
}

void nqueen (int k)
{
  int i,j;
  for(i=1; i<=n; i++)
    if ( place(k, i))
    {
      x[k]=i;
      if (k==n)
      {
        for(j=1; j<=n; j++)
          printf ("%d ", x[j]);
        printf("\n");
      }
      else nqueen(k+1);
    }
}
Void main()
{ printf ("Enter the number of queens\n");
  Scanf ("%d", &n);
  printf (" All posible solutions are :\n");
  nqueen (1);
}
```



★ GRAPH COLOURING PROBLEM :

- Given a graph of 'v' vertices and 'e' edges and 'n' different colours.
- We want to colour the graph with n colours in such a way that the adjustant vertices must not be of same colour.
- A graph is said to be 'm' colourable graph, if 'm' is the minimum no. of colour necessarily to colour all the vertices ensuring adjacent vertices are not of same colour.

- consider a graph :



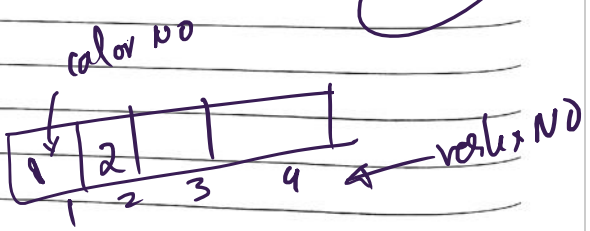
Here if we take 4 colors It is feasible as all vertex will be of diff color but not optimal because 4 is minimal value.

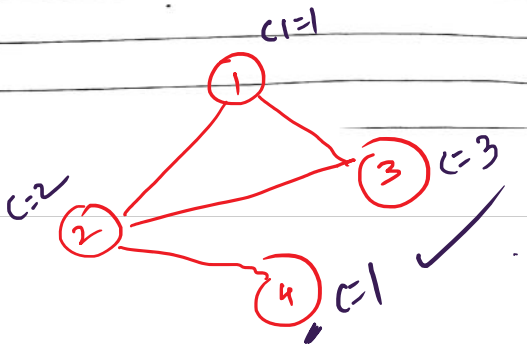
Problem? Again the graph have to minimum 3 the color we have change to ensure no two vertex are of same color adjacent

- The above graph is 3 colourable graph. Let, the colours be 1, 2 & 3.
- possible solutions are :

V_1	V_2	V_3	V_4
1	2	3	1
1	2	3	3
3	2	1	3
3	2	1	1
⋮			

m = 3





Solution :

Consider 1-D array x that has v elements where every element in x store color number.

x[j] = color NO
j = vertex NO.

Constrainst on array x :

(1) If i and j are adjacent vertices then x[i] != x[j]

(2) Every value in x must be between 1 to m

→ Algorithm :

Let g represents adjacency matrix of graph with v vertices and e edges

Let int x[v] stores solution.

Algo for color( )

[kth vertex pe ith color chalega kya ?]

```
int color (int k, int i)
{
    int j ;
    for ( j=1 ; j<=k-1 ; j++)
    {
        if ( g[k][j] !=0 && x[j] ==i)
            retum 0 ;
    }
    retum 1 ;
}
```

k : vertex number.

i : colour numbeg (proposed Color Number)

g[k][j] != 0 : kth vertex aur jth vertex adjacent hai kya

x[j] == i

x[j] → jth colour / vertex j

i → proposed colour of kth vertex

Algo. for graphcolor()

```
void graphcolor (int k)
{
    int i;
    for (i=1; i<=n; i++)
    {
        if (color (k,i))
        {
            x[k] = i;
            if (k==v)
            {
                for (j=1 to v)
                    print x[j]
            }
            else graphcolor (k+1);
        }
    }
}
```

k → Vertex No

chk for all possible n colors

agar Vertex k ko color i se color kar sakte hai

Toh color karo

kya ye last Vertex No hai

// solution.

call for next vertex.

* SUM OF SUBSET :→

Problem statement:

We are given 'n' positive integers and a positive integer and a number 's'.

We have to find all posible subset of 'n' such that their sum is 's'.

→ N = {1, 2, 3, 10, 12, 13, 15}

→ s = 15.

Sum req.

Here n=7

{15} ✓

{2,3,10} ✓

{3,12} ✓

Scanned by CamScanner

Possible solutions are:

$X = \{1 5\} \in N$

$X = \{2, 13\} \in N$

$X = \{3, 12\} \in N$

$X = \{1, 2, 12\} \in N$

$X = \{2, 3, 10\} \in N$

⋮

The solution for the problem can be found using Backtracking approach. Let their be 1-D array X that has n locations. The array X will store those integers whose sum will be S.

Constrains on array X[] are:

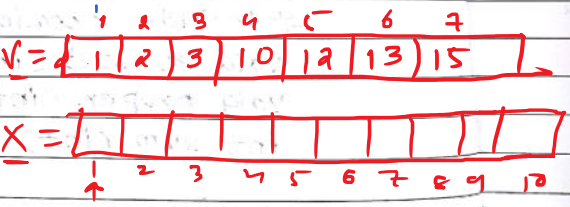
1. Sum of elements in array X should not exceed S. (Equal to S.)
2. The values stored in array x should be unique. as we do not want to repeat the subset.

Program:

```
#include <stdio.h>
#include <conio.h>
 void sumset(int);
 void proper (int,int) ;
 int sum (int)

 int v[10] ;          // To store elements
 int x[10] ;          // To store solution.
 int n, sum_req ;
```

```
void main()
{
 int i;
 clrscr();
 printf("Enter number of elements\n");
 scanf("%d",&n);
 printf("Enter unique elements in increasing order\n");
    for(i=1; i<=n; i++)
        scanf("%d",&v[i]);

 Printf("Enter sum required\n");
 scanf("%d",&sumreq);
 sumset(1);
 getch();
}
```

```
int sum(int n)                    // gives sum of first n elements
{                                    of array x.
  int s,i;
  s=0;
  for(i=1; i<=n; i++)
     s = s + x[i];
  return s;
}
```




```
                  1          1
int proper(int K, int i)
{                 ↑ Array X main K^th pos" par array V Ka i^th pos"
                    ka element dalega kya.
  int j;
  for(j=1; j<=K-1; j++)          // If same element is present before
  {   if(x[j]==v[i])             // Repeated toh nahi
         return 0;
}
```

```
        if ((sum(k-1) + v[i]) -> sumreq)   // purane kth element
            return 0;                          ka sum.
        else return 1;
    }

    int sumset (int k)
    {
        int i, s, j;
        for (i=k; i<=n; i++)
        {
            if (proper(k,i))
            {
                x[k] = v[i];
                s = sum(k);
                if (s == sumreq)
                {                              // Print solution
                    for(j=1; j<=k; j++)
                        printf("%d", x[j])
                }
                else
                    sumset(k+1);
            }
        }
    }
```

NOTE:
k is position in array x.
i is position in array v.
Proper() specifies whether element at ith element position can be placed at kth position in array x.

Possible solutions are:

$X = \{1 5\} \in N$

$X = \{2, 13\} \in N$

$X = \{3, 12\} \in N$

$X = \{1, 2, 12\} \in N$

$X = \{2, 3, 10\} \in N$

⋮

The solution for the problem can be found using Backtracking approach. Let their be 1-D array X that has n locations. The array X will store those integers whose sum will be S.

Constrains on array X[] are:

1. Sum of elements in array X should not exceed S. (Equal to S.)
2. The values stored in array x should be unique. as we do not want to repeat the subset.

Program:

```
#include <stdio.h>
#include <conio.h>
 void sumset(int);
 void proper (int,int) ;
 int sum (int)

 int v[10] ;          // To store elements
 int x[10] ;          // To store solution.
 int n, sum_req ;
```

```
void main()
{
 int i;
 clrscr();
 printf("Enter number of elements\n");
 scanf("%d",&n);
 printf("Enter unique elements in increasing order\n");
    for(i=1; i<=n; i++)
        scanf("%d",&v[i]);

 Printf("Enter sum required\n");
 scanf("%d",&sumreq);
 sumset(1);
 getch();
}
```

no of elements in array X

```
int sum(int n)                    // gives sum of first n elements
{                                    of array X.
  int s,i;
  s=0;
  for(i=1; i<=n; i++)
     s = s + x[i];}
  return s;
}
```

pos^n in array V

	1	2	3	4	5	6	7
V =	1	2	3	10	12	13	15

X =

	1	2	3	4	5	6	7	8	9	10

```
int proper(int k, int i)
{                  ↑ Array X main K^th pos^n par array V ka i^th pos^n
  int j;              ka element dalega kya.
  for(j=1; j<=k-1 ; j++)   // If same element is present before
  {   if (x[j]==v[i])      // Repeated toh nahi
        return 0;
}
```

Kya hum array V main se position i ka element array X main pos^n k pe rakh sakte hain kya.

purane element ↓ naya element

```
if ((sum(k-1) + v[i]) -> sumreq)   // purane K^th element
    return 0;                         ka sum.
else return 1;   → allowed.
}

int sumset(int k)
{                                  → n element in array V.
```

```
int sumset (int k)
{
    int i, s, j ;
    for (i=1 ; i<=n ; i++)
    {
        if (proper(k,i))
        {
            x[k] = v[i] ;
            s = sum (k);
            if ( s == sumreq)
            {
                for(j=1 ; j<=k; j++)
                    printf("%d", x[j])
            }
            else
                sumset (k+1) ;
        }
    }
}
```

n element in array v.

place ith element of v at posn k in array x.

// Print solution

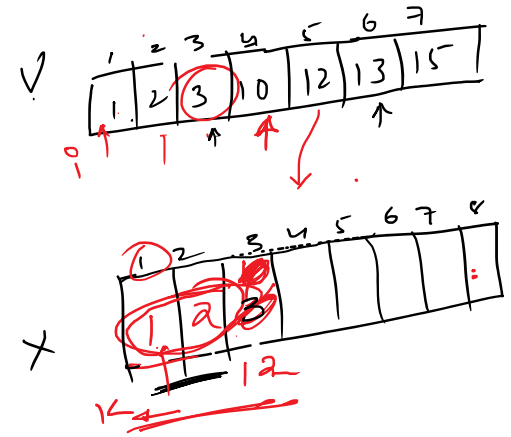
Ab array x main k+1 posn pe element karo

main()
{
sumset(1);
}

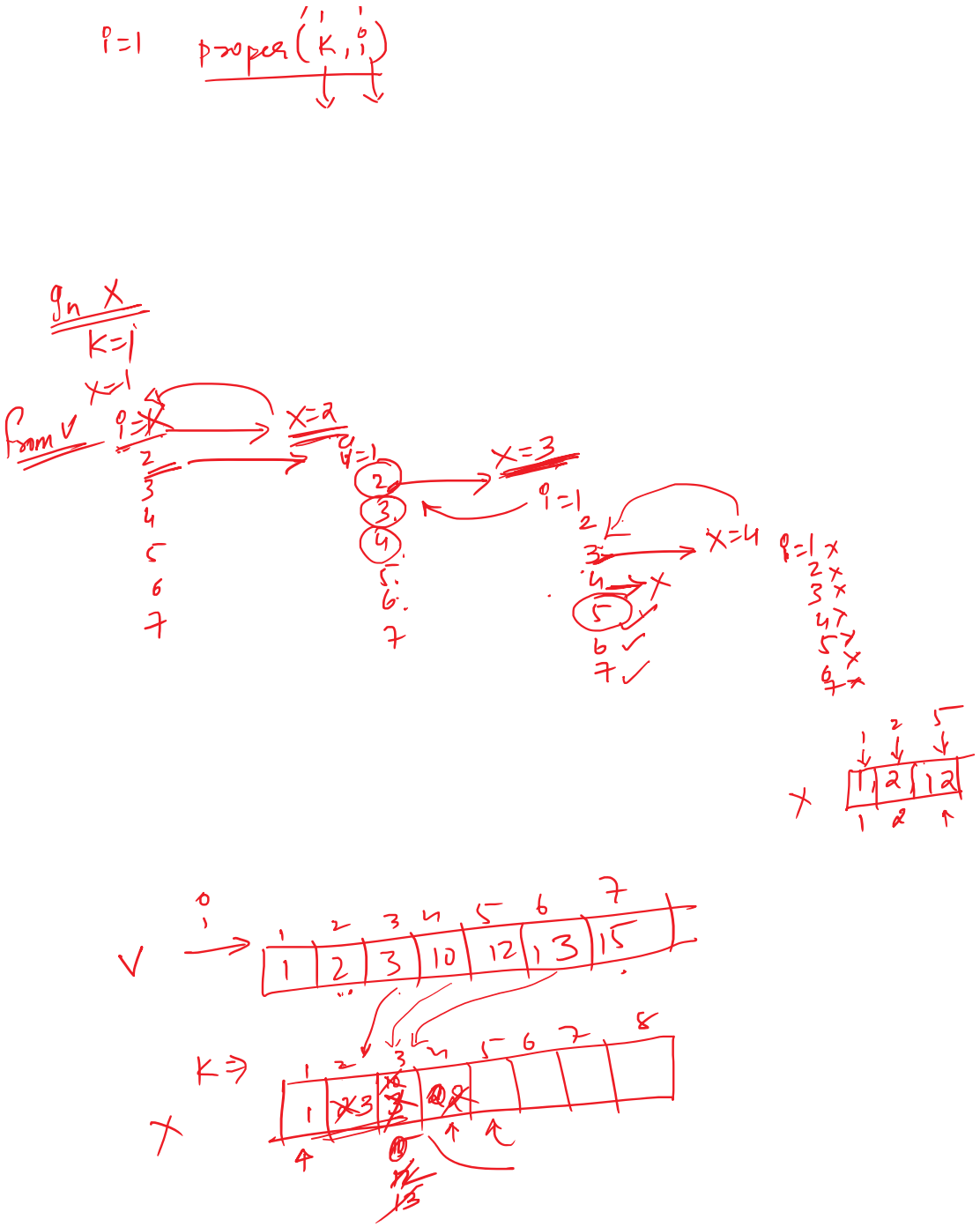
NOTE :
- k is position in array x. (solution)
- i is position in array v (input)

Proper () specifies whether element at ith element position can be placed at kth position in array x.


```
proper(int k, int i)
{
    for(j=1 to k-1)
    {
        if(x[j] == v[i])
            return 0;
    }
    if ( sum(k-1) + v[i] > sumreq)
        return 0;
    else return 1;
}
```



sum = 15

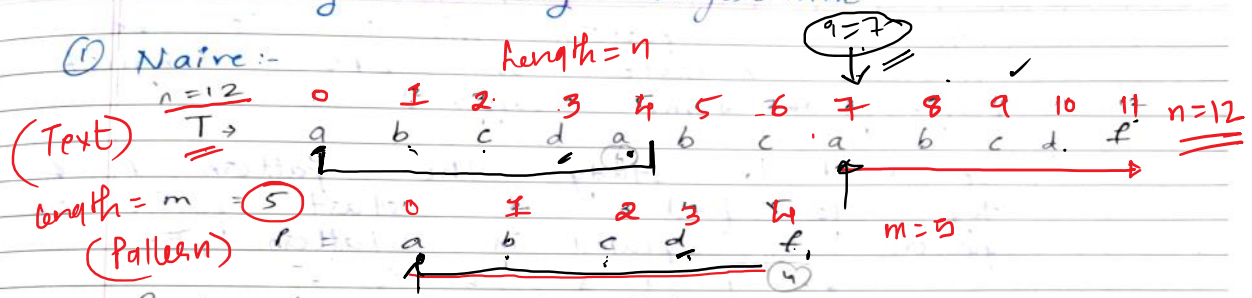

p=1
p=proc(k,i)
In X
k=1
X=1
from V
X=2
X=3
X=4

V
K⇒
X

It returns portion in Text T where pattern P appears.

String matching Algorithm

(1) Naive :-



Brute force (Naive) Pattern matching

i = iterator for Text
j = iterator for Pattern

Algorithm :

1. start
2. for $i = 0$ to $n - m$ do — (this is the last pos^n in Text T where P can appear.)
   {
3. $j = 0$
4. while $j \leq m$ and $P[j] = T[i+j]$ do
5. $j = j + 1$
6. if $j = m$ then
7. return i
   }
8. return -1

Complexity = $O(n)$

i=0, j=0: $P[0] = T[0+0]$ | q=0, i=0, j=1: $P[1] == T[0+1]$ | q=0, j=2: $P[2] = T[2]$

j=3: $P[3] = T[0+3]$ | i=4: $P[4] = T[0+4]$

(2) Knuth Morris Pratt (KMP) Pattern matching

Here, we study the pattern so we reduce the time needed. and it is not require backtrack again & again. The idea of KMP is to find within string if any prefix same as suffix. i.e is the beginning part of the pattern appears

again in pattern. So for this we generate π-table or LPS (longest Prefix for that some suffix)

ote:- We need to study the pattern P & try to find it start position of pattern is repeated in pattern part.

	0	1	2	3	4	5	6
Patt	a	b	c	d	a	b	c
π-table	0	0	0	0	1	2	3

	a	b	c	d	a	b	e	a	b	f
π-table	0	0	0	0	1	2	0	1	2	0

	a	b	c	d	e	a	b	f	a	b	c
π-table	0	0	0	0	0	1	2	0	1	2	3

	a	a	b	c	a	d	a	a	b	c
π-table	0	1	0	0	1	0	1	2	3	0

	a	a	a	a	b	a	a	c	d
π-table	0	1	2	3	0	1	2	0	0

Algo to prepare π-table for given Pattern P

KMP_failure (P)

P(1 : m) = Pattern

f(1 : m)

1. start
2. f(0) = 0 , i = 1 , j = 0
3. while (i < m) do
{
4. if (p(j) = p(i)) then
5. f(i) = j + 1
6. i = i + 1 , j = j + 1
}
7. else if (j > 0) then
8. j = f(j-1)
9. else
{
10. f(i) = 0
11. i = i + 1 ;
}
12. return

Consider,

	0	1	2	3	4	5	
p =	a	b	a	c	a	b	length m = 6

i	j	f(0)	f(1)	f(2)	f(3)	f(4)	f(5)
1	0	0	0				
2	0			1			
3	1						
3	0			0			
4	0				1		
5	1						2
6	2						

```
function KMP_Match(T,P)
    T(1:n) , P(1:m)
    Integer f(1:m)
1. start
2. i=0      j=0
3   while ( i<n )  do
    {
    if ( P(j) == T(i)) then
    {
            if (j == m-1) then
                return  i-m+1
            else
                i=i+1 , j=j+1
    }
    else if ( j>0 ) then  j = f(j-1)
    else  i=i+1
}
```

Consider P =

	a	b	a	c	a	b
	i	i	i	i	i	i
	j					

T =

	0	1	2	3	4	5	6	7	8	9	10
	a	b	a	c	b	a	b	a	c	a	b
	i	i	i	i	i	i					

f =

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

At i = 4 & j = 4 , P(j) ≠ T(i)

∴ put j to f(j-1)

Here f(4-1) = f(3) = 0

∴ j = 0

Hindi main

i & j ko 0 se start karo

Jabtak (P(j) == T(i))

{

i++ , j++

Agar j = m-1 (last character) mila at i-m+1

}

Agar [P(j) != T(i)] to j ko peeche le jao

Kahan j = f(j-1)

Agar j Already 0 hai toh i ko agge le jao i = i+1

T = a b a b c a b c a b a a b a b d

P = a b a b d

f = 0 0 1 2 0

3) Robin Karp:

- It finds a pattern within a text using hashing
- It is used for finding multiple occurrence of the pattern in the text

Algorithm:-

fn Robin_Knap (Txt , pat , d , q)

Txt (1:n) → text

pat (1:m) → pattern

d → length of i/p alphabet (d = 256)

q → prime number

```
1. Start
2. p = 0 , t = 0
3. h = d^(m-1) mod q
4. for i = 0 to m-1 do
   {
5.     p = (d * p + Pat(i)) mod q
6.     t = (d * t + Txt(i)) mod q
   }
7. for i = 0 to n-m do
   {
8.     if p = t then
       {
9.         for j=0 to m-1 do
           {
10.            if Txt(i+j) ≠ Pat(j) then
11.                go to step 12
           }
12.        if j = m then
13.        print " Pattes found at ", i
       }
14. if i < n-m then
    {
15.     t = [d * (t - Txt(i) * h) + Txt(i+m)] mod q
16.     if t < 0 then
17.         t = t + q
    }
18. return.
```

a b c d
43 98 99 (100)

Consider

n = 15

Text = [t|o|d|a|y| |i|s| |s|u|n|d|a|y]
0 1 2

m = 3

Pat = [d|a|y]

a = 97
d = 256
q = 101

$h = 256^2 \bmod 101 = 88$

h = 88

i = 0 to 2

i = 0, p = 0 t = 0

i = 0

p = (d * p + Pat(i)) mod q
= (256 * 0 + 'd' 100) mod 101
= 100

t = (256 * 0 + 't' 116) % 101 = 15

i = 1

p = (256 * 100 + 'a' 97) % 101 = 43
t = (256 * 15 + 'o' 111) % 101 = 12

i = 2

p = (43 * 256 + 'y' 121) % 101 = 19
t = (256 * 12 + 'd' 100) % 101 = 41

i = 0, p = 19, t = 41

∵ p ≠ t

Calculate Next t

t = [256 * (41 − 't' 116 * 88) + 'a' 97] % 101 = −87

∵ t < 0

∴ t = −87 + 101

t = 14

i = 1, p = 19, t = 14

p t t

Nxt t t

t = (256 * (14 - 'o' 111 * 88) + 'y' 121) % 101

t = -82

t < 0

t = -82 + 101

t = 19

i = 2 p = 19 t = 19

∴ p = t

Check pattern

j = 0 'd' 'd'

j = 1 'a' 'a'

j = 2 'y' 'y'

found

KMP (Knuth Morris Pratt) Algorithm for Pattern matching

Basic Idea of working

Pattern ⇒ a b c d a b c

Prefix ⇒ a, ab, abc, abcd, abcda, abcdab, abcdabc

(Pattern from start)

Suffix → c, bc, abc, dabc, cdabc, bcdabc, abcdabc

(subset from last)

The basic idea of KMP is to check if Inside a String is any Prefix Same as Suffix

[KMP will if pattern from beginning of text (prefix) appears again in the end (suffix)]

Here we generate π table to get LPS (longest Prefix which is Same as Suffix)

Consider P_1 = a b c d a b e a b f

π = 0 0 0 0 1 2 0 1 2 0

π =

P_2 =

1	2	3	4	5	6	7	8	9	10	11
a	b	c	d	e	a	b	f	a	b	c

π =

0	0	0	0	0	1	2	0	1	2	3

P_3 =

1	2	3	4	5	6	7	8	9	10
a	a	b	c	a	d	a	a	b	e

π

0	1	0	0	1	0	1	2	3	0

Consider an Example to find occurance of P in T ⇒

T =

i 1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	b	a	b	c	a	b	c	a	b	a	b	a	b	d

P =

d=0 1	2	3	4	5
a	b	a	b	d

Let i starts at posn 1 in T

Let j starts at pos 0 in P (till will make simple)

KMP Approach ⇒ Compare T[i] with P[j+1]

if matches then increment i and j both

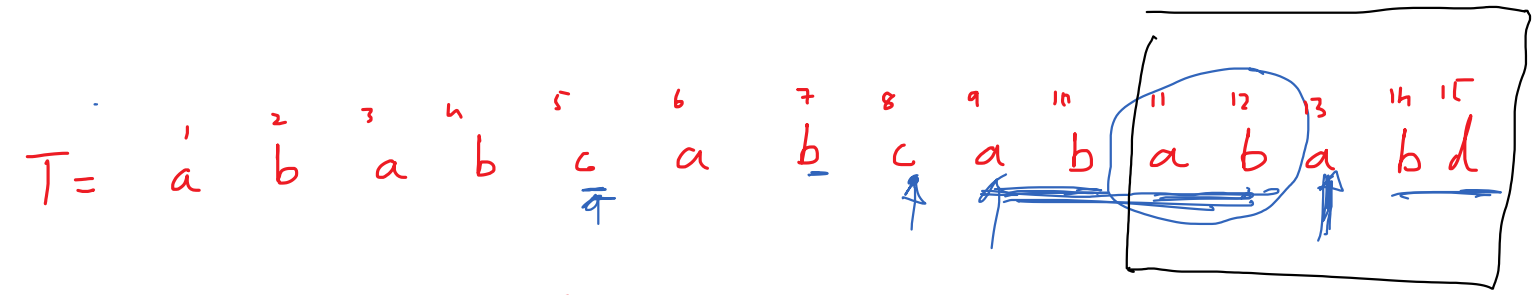
if there is mismatch betn T[i] and P[j+1]

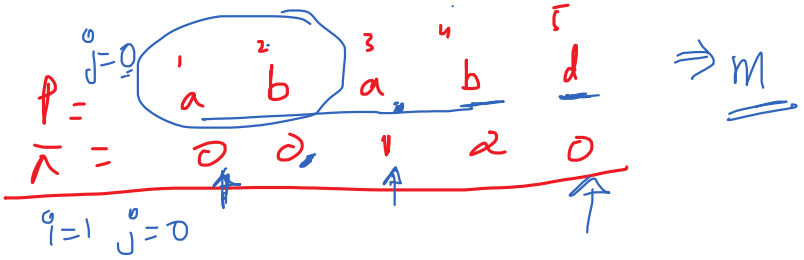
then move j to π[j] and again math

T[i] with P[j+1]

But if j is already 0 so cannot move back in

But if j is already 0 so cannot move back ...
Such case move i = i+1 and again
match T[i] and P[j+1]





⇒ m

T[i] == P[j+1] = P[0+1]

T[1] = P[1] Yes i++, j++

T[2] = P[2] Yes i++, j++

T[3] = P[3] Yes i++, j++

T[4] = P[4] Yes i++, j++

T[5] = P[5] NO

Now j goes to π(j) ie π(4)

π(4) = 2

∴ i=5, j=2

Is T[5] = P[j+1] = P[3] NO

j will go to π(j), ie π(2)

π(2) = 0

So j=0

Now i=5, j=0

Is $T(5) == P(j+1) = P(0+1) = P(1)$ NO

But j is already 0

Now move i to $i+1$

$i = i+1 = 5+1 = 6$

$j = 0$

Again $T[i] == P(j+1)$?

$i = 6, \; j = 0$

Is $T(6) == P(1)$ Yes $i++, j++$

$T(7) == P(2)$ Yes $i++, j++$

$T(8) == P(3)$ NO

j moves to $\pi(j)$ i.e $\pi(2)$

$\pi(2) = 0$

$\therefore j = 0, \; i = 8$

Check if $T(8) == P(j+1) = P(0+1) = P(1)$ NO

Now move j to $\pi(j)$ but j is already 0

So move i to $i+1$

$\therefore i = i+1 = 8+1 = 9 \qquad j = 0$

$\therefore i = 9, \; j = 0$

Is $T(9) == P(0+1) = P(1)$ Yes $i++ \; j++$

$T(10) = P(2)$ Yes $i++, j++$

$T(11) = P(3)$ Yes $i++, j++$

$T(12) = P(4)$ Yes $i++ \; j++$

$T(13) = P(5)$ NO

move j to $\pi(j)$ i.e $\pi(4)$

i.e $\pi(4) = 2$

- move j to 2

$i = 13$, $j = 2$

Is $T(13) == P(j+1) = P(2+1) = P(3)$ Yes i++, j++

$T(14) == P(4)$ Yes i++ j++

$T(15) = P(5)$ Yes

$i = 15$ $j = 4$ Pattern found at $i - j = 15 - 4 = 11$

Knuth Morris Pratt ⇒

Here we study the Pattern so we reduce the time needed. and it is not required to backtrack again

The idea of KMP is to find within string if any

Prefix is same as suffix.

(if the beginning of pattern appears in later part).

→ We will generate π table or LPS (longest Prefix for Some Suffix)

* Note ⇒ we need to study the pattern P

i.e we will try to find if start pos^n of pattern is repeated in later part

i	0	1	2	3	4	5	6
P =	a	b	c	d	a	b	c
f(i) = π(i)	0	0	0	0	1	2	3

i ⇒	0	1	2	3	4	5	6	7	8	9
P	a	b	c	d	a	b	e	a	b	f
f(i) = π(i)	0	0	0	0	1	2	0	1	2	0

P	a	b	c	d	e	a	b	f	a	b	c
f(i) = π(i)	0	0	0	0	0	1	2	0	1	2	3

i ⇒	0	1	2	3	4	5	6	7	8	9
P	a	a	b	c	a	d	a	a	b	e
f(i) = π(i)	0	1	0	0	1	0	1	2	3	0

Algo to Prepare π table for given Pattern.

```
kmp_failure (P) // π function.
{  P(1:m) : Pattern
   f(1:m) : π function.

1. start.
2. f(0)=0, i=1, j=0.
3. while (i<m) do
   {
4.    if(P(j) = P(i)) then
      {
5.       f(i) = j+1
6.       i = i+1;
7.    } j = j+1;
8. else if(j>0) then
9.       j = f(j-1);
10. else
    {
11.      f(i)=0;
12.      i=i+1;
    }
13. return;
}
```

Tarikan 1

i	0	1	2	3	4	5
P =	a	b	a	c	a	b
π(i)=f(i)=	0	0	1	0	1	2

Tarikan 2 Algo. m = 6

P =

i	j	f(0)	f(1)	f(2)	f(3)	f(4)	f(5)
		a (0)	b (1)	a (2)	c (3)	a (4)	b (5)
1	0	0					
2	0		0				
3	1			1			
3	0				0		
4	0					1	
5	1						2
6	2						

m = 6

i ⇒	0	1	2	3	4	5
P =	a	b	a	c	a	b
f(i) = π(i) ⇒	0	0	1	0	1	2

```
function kmp_match( T, P)
{  T(1:n) ⇒ Text
```

n = 11

i	0	1	2	3	4	5	6	7	8	9	10
T =	a	b	a	c	b	a	b	a	c	a	b

function KMP_Matcher ...

{ T(1:n) ⇒ Text
P(1:m) ⇒ Pattern
f(1:m) ⇒ π function

1. start
2. i=0, j=0
3. while(i<n) do
{
if (P(j) == T(i)) then.
{ if (j = m-1) then
return i-m+1;
else
{ i=i+1;
j=j+1; ✓
}
else if(j>0) then
j=f(j-1);
else
i=i+1;
}

T = a b a c b a b a c a b

i=0	j=0
1	1
2	2
3	3
4	4
	j=f(j-1)
	f(3)
4	=0
5	0

10-6+1
= 4+1 = 5

Hindi main ⇒

i & j ko bolo 0

Jabtak (P(j) == T(i))

d i++, j++

Agar j == m-1

reborn mila at i-m+1 posn.

}

Agar (P(j) != T(i)) to j ko Peeche leke jaao

j = f(j-1)

Agar j already 0 hai toh i = i+1.

Rabin Karp Algo for String Matching :->

* It finds a pattern within a text using hashing
* It is used to determine multiple occurances of pattern in the Text.

Algorithm :->

function Rabin_Karp (Txt, Patt, d, q)

{ Txt(1:n) -> Text
Patt(1:m) -> Pattern.
d = length of input dataset. (d = 256) -> base.
q = prime No.

1. start
2. p = 0 , t = 0 ✓
3. h = d^{m-1} mod q ✓
4. for (i = 0 to m-1) do
5. { p = (d * p + Patt(i)) mod q.
6. t = (d * t + Txt(i)) mod q
   } ✓
7. for (i = 0 to n-m) do
   {
8. if (p == t) then -> if Hash value matched then
   {
9. for (j = 0 to m-1) do
   { if (Text(i+j) ≠ Patt(j)) then -> character by character matching
10.
11. go to step 12
   }
12. if (j == m) then
13. print " Pattern found at " + i.
   }
14. if (i < n-m) then.
   {
15. t = (d * (t - Text(i) * h) + Text(i+m)] mod q
16. if (t < 0) then

```
16      if (t<0) then
17          t = t+q.
        }
18  }
19  return.
```

Let n=15

Text =

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
t	o	d	a	y		q	s		s	u	n	d	a	y

m=3.

Pattern =

0	1	2
d	a	y

d=256.

q = 101 (prime Number). → Given/Assume

Sol p=0, t=0

$h = d^{m-1} \bmod q = 256^2 \bmod 101$

$h = 88$

Now for (i=0 to 2)

{ d p = (d*p + Patt(i)) mod q

t = (d*t + Txt(i)) mod q }

}.

i=0

$p = (256*0 + 'd') \bmod q$ ('d' = 100)

$= (256*0 + 100) \bmod 101$

$= 100$

$t = (256*0 + 't') \bmod 101$ ('t' = 116) → ASCII value.

$= (0 + 116) \bmod 101$

$= 15$

i=1 $p = (256*100 + 'a') \bmod 101 = (256*100 + 97) \bmod 101 \Rightarrow 43$ ('a' = 97)

$t = (256*15 + 'o') \bmod 101 = (256*15 + 111) \bmod 101 \Rightarrow 12$. ('o' = 111)

i=2

$p = (256*43 + 'y') \bmod 101 = (256*43 + 121) \bmod 101 \Rightarrow 19$ ('y' = 121)

$t = (256*12 + 'd') \bmod 101 = (256*12 + 100) \bmod 101 \Rightarrow 41$. ('d' = 100)

After for loop. p=19, t=41

Now for (i=0 to 12) — for. i=0

if p==t ? No

Hence p≠t

calculate Next 't'

$t = (d * (t - Text(i) * h) + Text(i+m)) \bmod q$.

$= (256 * (41 - 't' * 88) + Text(0+3)) \bmod q$ ('t' = 116)

$= (256 * (41 - 't'^{116} * 88) + Text(0+3)) \bmod q$

$= (256 * (41 - 116 * 88) + 'a'^{97}) \bmod 101$

$= -87$

Since $t = -87 < 0$

$\therefore t = t + q = -87 + 101 = 14$

for $i=1$, $p=19$, $t=14$.

Again $p \neq t$

Calculate next t.

$t = (256 * (14 - 'o' * 88) + Text[1+3]) \bmod q$

$= (256 * (14 - 111 * 88) + 'y'^{121}) \bmod 101$

$= (256 * (14 - 111 * 88) + 121) \bmod 101$

$t = -82$

Since $t < 0$

$t = t + q$

$= -82 + 101$

$t = 19$

for $i=2$, $p=19$, $t=19$

Since $p = t$ ✓

```
if (p==t) then
  for (j=0 to m-1) do
    if (Text(i+j) ≠ Patt(j)) then
      go to step 12
  if (j==m) then
    print "Pattern found at" i
```

text = t o d a y i s sunday
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

Patt = d a y
0 1 2

$i=2$

$j=0$ to 2

$j=0$ Text(2+0) == Patt(0) Yes

$j++$ $j=1$ Text(2+1) == Patt(1) Yes

$j++$ $j=2$ Text(2+2) == Patt(2) Yes.

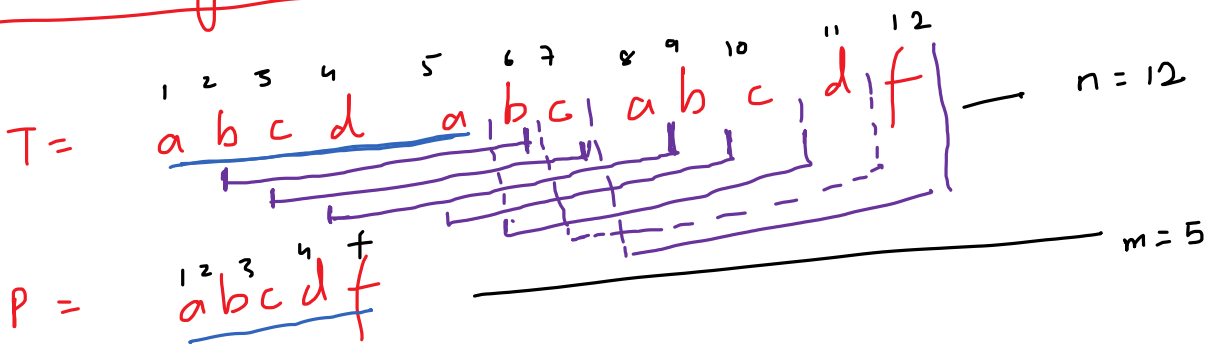
$j=3$ $j<2$ no

Here $j=3==m$

Pattern found at 'i' ⇒ (2)

(Not) for Exam

Shortcomming in Naive / Brute force Approach



Here in case of Mismatch the window of Pattern Search in Text T is moved by 1 position.

Robin Karp Pattern Matching Algorithm →

Approach Implemented

Consider →

Text : a a a a a b ⇒ n = 6

P : a a b m = 3

* Here character by character matching is not Implemented

* Here the idea is to Convert the pattern into single numeric value

Let character set →

a = 1	f = 6
b = 2	g = 7
c = 3	h = 8
d = 4	⋮ = 9
e = 5	j = 10

1 way ⇐ Using this

P = | 10^2 | 10^1 | 10^0 |
|---|---|---|
| a | a | b |

$P = \{ a\ a\ b \}$ $10^2\ 10^1\ 10^0$

$1\times10^2 + 1\times10^1 + 2\times10^0$

$\Rightarrow 100 + 10 + 2 = 112$ ✓

$h(P) = 112$

Let there be a base = 10 [10 character]

$c = \leq\ i, j < 10$

Consider Text. a a a a a b (1 2 3 4 5 6)

$1\times10^2 + 1\times10^1 + 1\times10^0$

$= 111$

Not same as $h(P)$

$h(T(1-3)) \neq 111$

$h(T(2-4)) = a\ a\ a$ $1\times10^2 + 1\times10^1 + 1\times10^0$ $\Rightarrow 111 \neq h(P)$

$h(T(3-5)) \Rightarrow a\ a\ a$ $1\times10^2 + 1\times10^1 + 1\times10^0$ $\Rightarrow 111 \neq h(P)$

$h(T(4-6)) = a\ a\ b$ $1\times10^2 + 1\times10^1 + 2\times10^0$ $\Rightarrow 112 = h(P)$

Once the hash value matches, now match the $T(4-6)$ with Pattern

Consider -> $T =$ c c a c c a a e d b a (1 2 3 4 5 6 7 8 9 10 11)

$P =$ d b a (1 2 3)

a = 1
b = 2
c = 3
d = 4

e = 5

Let the hash f^n be Sum of code values of characters ⇒.

$h(P) = \underset{4+2+1}{d\ b\ a} \Rightarrow 7$

T ⇒ c c a c c a a e d b a

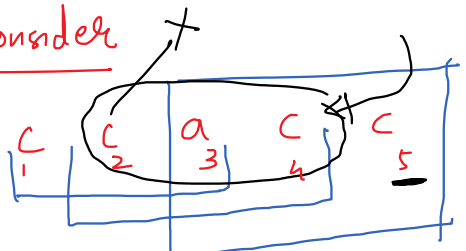
$3 + 3 + 1 = 7$

Here Even though the hash value matches but the text does not matches.

* These match are known as "spurious hits"
* Our goal is to minimize spurious hits
* In above case we have spurous hits because the hash f^n is very simple
* There is need to have good hash function that minimizes spurous hits

* Rabin Karp Suggest Such fundion

Now Consider



P = d b a
1 2 3

m = 3

$h(P) = d\ b\ a$

let
a = 1
b = 2
c = 3
d = 4
e = 5
f = 6

$h(P) = d\ b\ a$ $\quad m=3$

$4\times10^2 + 2\times10^1 + 1\times10^0 \Rightarrow 421$

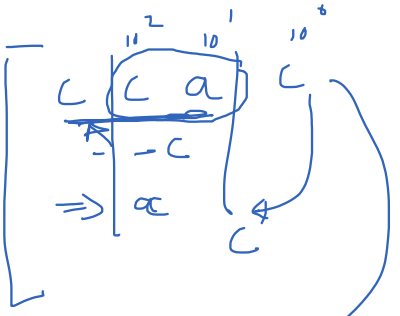
f-6
g-7
h-8
:
j 10

Base=10

Apply same hash fn on first m characters of Text T.

$h(T(1-3)) = c\ c\ a \Rightarrow 300+30+1 = 331 \neq h(P)$

$3\times10^2 + 3\times10 + 1\times10^0$



$h(T(2-4)) = (3\times10^2 + 3\times10 + 1\times10^0) - 3\times10^2$ $\quad$ cca

$= 3\times10^1 + 1\times10^0$ $\quad$ ca

$\Rightarrow (3\times10^1 + 1\times10^0)\times10$

$\Rightarrow 3\times10^2 + 1\times10^1 + 3\times10^0$

$\Rightarrow 300 + 10 + 1 = 311 \neq h(P)$

$h(T(2-4)) \Rightarrow c\ a\ c$

$3\times10^2 + 1\times10^1 + 3\times10^0$

$h(T(3-5)) = ((3\times10^2 + 1\times10^1 + 3\times10^0) - 3\times10^2)\times10 + 3\times10^0$

(a) 3 (c) 4 (c) 5

$= 1\times10^2 + 3\times10^1 + 3\times10^0 \Rightarrow 100 + 30 + 3 = 133$

$\neq h(P)$

Non-deterministic Polynomial Algorithms

Polynomial Time

An algorithm is said to be solvable in polynomial time if the number of steps required to complete the algorithm for a given input is O(n^k) for some nonnegative integer k, where n is the complexity of the input.

Polynomial-time algorithms are said to be "fast."

Most familiar mathematical operations such as addition, subtraction, multiplication, and division, as well as computing square roots, powers, and logarithms, can be performed in polynomial time.

Computing the digits of most interesting mathematical constants, including pi and e, can also be done in polynomial time.

Polynomial Time Verification

Given a problem instance and solution (certificate), the process of verifying in polynomial time whether the given solution solves the problem or not is called polynomial time verification.

P, NP and NPC Classes

P Class – The class P consists of those problems that are solvable in polynomial time. Specifically, they are the problems that can be solved in time $O(n^k)$ for some constant k, where n is the size to the input problem.

NP Class – The NP class consists of those problems that are verifiable in polynomial time. It means that if we were given a certificate of a solution, then we would verify that the certificate is correct in polynomial time of the size of the input.

Any problem in P is also in NP, since if the problem is in P then we can solve it in polynomial time without using certificate.

NPC Class – The problem is in NP Complete (NPC) class, if it is in NP and is as hard as any problem in NP.

If any NP Complete problem has polynomial time solution then all NP Complete problem has polynomial time solution.

Most of the theoretical computer scientists consider NP Complete problems to be intractable, since given the wide range of NP Complete problems studied till date without anyone having discovered a polynomial time solution to any of them.

① If it is NP
② As hard as any problem in NP

Ex Vertex is NPC?

① Proove vertex cover is NP. → Easily.

②

Reducibility

Let us consider a decision problem A which we would like to solve in polynomial time.

Now suppose there is a different decision problem B, that we already know how to solve in polynomial time.

Suppose that we have a procedure that transforms any instance (input) x of A into some instance y of B with the following characteristics.

(1) The transformation takes polynomial time.

(2) The answers are the same. That is, the answer of x is 'yes' if and only if answer of y is also 'yes'.

Such a procedure is called 'polynomial time reduction algorithm' and it provides a way to solve problem A in polynomial time:

(1) Given instance x of problem A, use the polynomial time reduction algorithm to transform it to instance y of problem B.

(2) Run the polynomial time decision algorithm for B on the instance y.

(3) Use the answer for y as answer for x.

As long as each of these steps takes polynomial time, all these together would also take polynomial time.

So by reducing solving problem A to solving problem B, we have used the easiness of B to prove the easiness of A.

Taking the idea further, we could use polynomial time reductions to show that no polynomial time algorithm can exist for particular problem B.

Suppose we have a decision problem A for which we already know that no polynomial time algorithm can exist.

Further we have a polynomial time reduction algorithm transforming instances of A to instances of B.

Now a simple proof by contradiction can be used to show that no polynomial time algorithm can exist for problem B.

Suppose B has a polynomial time algorithm, then using the above method we can show that problem A also has polynomial time algorithm.

But then it contradicts with our basic assumption that there is no polynomial time algorithm for problem A.

NP Completeness

Polynomial time reductions provide formal means of showing that one problem is at least as hard as another, to within a polynomial time factor.

Now we can define a set of NP Complete problems.

A problem X is said to be NP complete if

(1) X belongs to NP

(2) Every other problem Y of NP can be reduced to X in polynomial time.

If the problem satisfied the property 2 but not necessarily property 1 then the problem is said to be NP-hard.

NP Completeness Proofs

The method for proving that the given problem is NP complete is as follows:

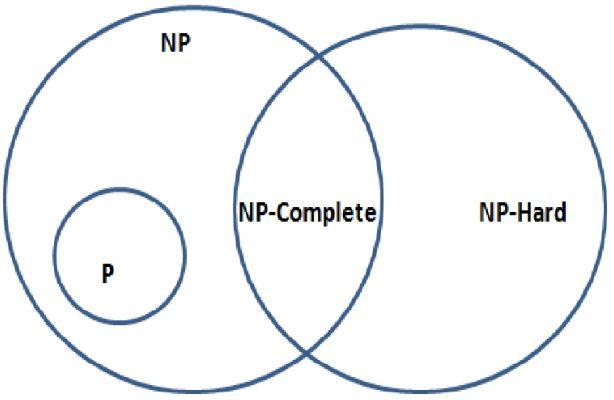
(1) Prove the given problem X belongs to NP.

(2) Select a known NP complete problem Y.

(3) Describe an algorithm that computes function f mapping every instance x ϵ {0,1}* of problem Y to an instance f(x) of problem X.

(4) Prove that the function f satisfies x ϵ Y if and only if f(x) ϵ X for all x ϵ { 0, 1 }*.

(5) Prove that algorithm computing f runs in polynomial time.



Consider

Polynomial Time.

Linear Search = $O(n)$

Binary search = $O(\log n)$

Insertion Sort < Best $\Omega(n)$ / Worst $O(n^2)$

Bubble Sort < Best $\Omega(n^2)$ / Worst $O(n^2)$

Exponential Time.

Sum of Subsets = 2^n

Graph Colouring = 2^n.

Future work → Do research to try To reduce complexity.

for Polynomial Time —Aim for→ Constant

for Exponential Time —Aim for→ Polynomial Time.

For CNF Satisfiability →

Consider $CNF = (x_1 \vee \overline{x}_2 \vee x_3) \wedge (\overline{x}_1 \vee x_2 \vee \overline{x}_3)$

Let $x_i = \{x_1, x_2, x_3\}$.

We want to find value of $x_1\ x_2\ x_3$ that satisfies above

x_1	x_2	x_3
1	1	1
		0
1	0	
0	1	

2^3

If we have 3 variable $\Rightarrow 2^3$ possibilities

If we have n variables $= 2^n$ possibilities $\Rightarrow$ Exponential time.

P $\Rightarrow$ Problems that can be solved in Polynomial Time.

NP $\Rightarrow$