LECTURE 21

→ DYNAMIC PROGRAMMING

(1) Coin exchange problem :-

given coins of denominations

$$1 = d_1 \le d_2 \le d_3 \le \ldots\ldots \le d_k$$

d_k = infinite # of each type.

otherwise, I cannot pay 'n' always.
eg: $d_1 = 2$, $d_2 = 5$
then 8 XX
not possible!

We need to find the minimum # of coins required to pay for 'n'.

→ greedy algorithm does not always give correct answer!

→ eg: $d_1 = 1$, $d_2 = 10$, $d_3 = 25$.
pay ₹30.
greedy → 25, 1, 1, 1, 1, 1
optimal → 10, 10, 10

→ eg: $d_1 = 1$, $d_2 = 7$, $d_3 = 10$
pay ₹14
greedy → 10, 1, 1, 1, 1
optimal → 7, 7

→ eg:- ATM m/c, dispense ₹n, then minimum # of coins required to pay 'n'?

→

- The problem was introduced in 1950 by Bellman as a method for optimizing control systems.

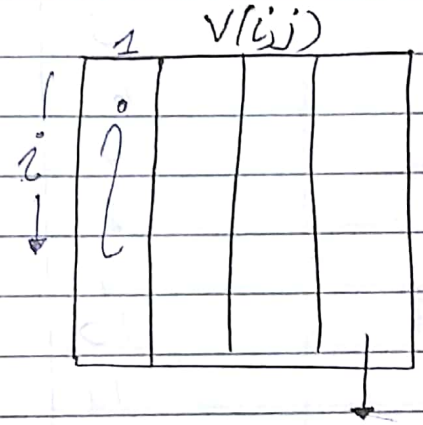
- definition:

given $d_1, d_2, d_3, \ldots d_k$
where $d_1 = 1$; and # of coins of denomination d_i are infinite.

find $V(i,j)$ = min. # of coins required to pay 'i' using first 'j' coins.

amount I pay for it

# of coins allowed



$V(i,1) = i$ coins.
eg:- $V(5,1) = 5$

so,

$\boxed{d_1, d_2, \ldots d_{i-1},} \; d_i, d_{i+1} \ldots d_k \quad V(n,k)$

look at the coins used before

I'm here

main idea :-

j^{th} coin

pick j^{th} coin

do not pick j^{th} coin

$1 + V(i - d_j, j)$

$V(i, j-1)$

# of coins goes up by one.

same as it was with $(j-1)$ coins

so, $V(i,j) = \min\{V(i, j-1), \underbrace{V(i-d_j, j)}_{i \geq d_j} + 1\}$

eg:- $d_1 = 1$, $d_2 = 7$, $d_3 = 10$.

i \ j	1	2	3
1	1	1	1
2	2	2	2
3	3	3	3
4	4	4	4
5	5	5	5
6	6	6	6
7	7	1	1
8	8	2	2
9	9	3	3
10	10	4	1
11	11	5	2
12	12	6	3
13	13	7	4
14	14	2	2
15	15	3	3

eg:- $V(14,2) = \min(\overset{14}{V(14,1)}, \underbrace{1 + \overset{1}{V(7,2)}}_{V(14,2) = 1+1 = \underline{2}})$

$\longrightarrow$

Code:-

```
for i = 1 to n
        V(i,0) = 0
        V(i,1) = i

for j = 1 to k
for i = 1 to n
        if (V(i,j-1) < (V(i-dj, j) +1))
                V(i,j) = V(i,j-1)
        else
                V(i,j) = 1+ V(i-dj, j)
```

} I can exchange/interchange these two terms, it does not matter.

final answer = V(n,k)

Time Complexity = O(nk)
Space Complexity = O(nk).

↳ what all coins did I use? ↓

```
i = n,   j = k
while (j > 1)
{
    if (V(i,j) == V(i,j-1))
            j = j-1;
    else {
            print("Used %d coins:", j);
            i = i-dj;
    }
}
Print("Pay by %d coins of type 1", i);
```

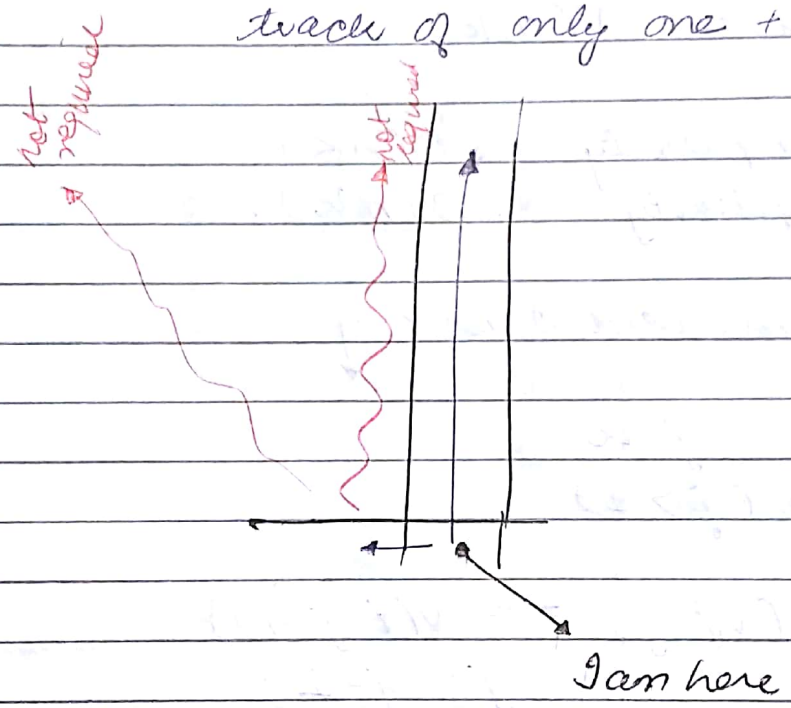
O(n) = O(n+k)

now, $O(nk)$ time is fine, but $O(nk)$ space is not?

eg: $n = 10^6$
$k = 10^3$

↳ We don't need the 1st column; I already need the entry from ~~some~~ previous row or some entry just above current column above it;

↳ so, at any time I need to keep a track of only one + one = two columns.



So, I can manage with only one column.

$V \longrightarrow n$ - length array.

$\longrightarrow$

```
for i = 1 to n
        V[i] = 1

for j = 2 to k
{ for i = 2 to n
    {
        if ( i >= d_j & (V(i-d_j)+1) < V(i))
                V(i) = V(i-d_j)+1
        else ;
    }
}
```

Time = O(nk)
Space = O(n)

↳ Here, I need backtracking to find what all coins I need. →

```
V(n) = minimum # of coins
  i = n,    j = k

while (j > 1)
{
    if (V(i) == 1 + V(i-d_j))
      { print ("j^th coin used");
            i = i - d_j;
      }
    else { j = j - 1; }
}
```

time complexity ~~remains~~ becomes ~~same~~ O(n+k)
but this doesnot work all the time!

↳ It is space minimization & not the time.

→ modified problem :-
know total # of ways to pay.

$$T(0, j) = 0$$
$$T(i, 1) = 1$$

$$\begin{tikzcd} & j \arrow[dl, "\text{select}"'] \arrow[dr, "\text{not select}"] & \\ T(i-d_j, j) & & T(i, j-1) \end{tikzcd}$$

$$T(i, j) = T(i-d_j, j) + T(i, j-1)$$

eg: $d_1 = 1$, $d_2 = 4$, $d_3 = 11$
pay 14 →
10, 1, 1, 1, 1
11, 1, 1, 1
4, 4, 4, 1, 1.

→ do I need 2nd table here or I can still manipulate 1D table?

↓

yes, I can still manipulate 1D table.

→ Steps to approach a DP problem :-
- characterize the structure of the optimal solution.
- recursively define the value of optimal solution.
- Compute the value of optimal solution in a bottom up fashion.

Compute an optimal solution from computed/stored information.

eg:- Fibonacci:-

$F[0] = 0$
$F[1] = 1$
for $i = 2$ to n
$f_i = (f_{i-1} + f_{i-2}) \% 10$.

tracks previous two values

space = Constant
time = $O(n)$.

→ 0/1 Knapsack :- (Repetition of item not allowed)

Given n-items,

int. weights	w_1	w_2	.....	w_n
values	v_1	v_2	.....	v_n

a knapsack of integer capacity 'w'
find most valuable subset of items that fit into the knapsack.

x_i = boolean variable.

x_i → 0 (not picking i^{th} item)
→ 1 (picking i^{th} item)

now, our objective is to find max value of $\Sigma x_i v_i$
given that condition
$\Sigma x_i w_i$ fits in the knapsack.

i.e. $\max(\sum x_i v_i)$ when $\sum x_i w_i < W$

this is an ILP (Integer linear Program) and an NP Hard Problem.

eg:-

item	wt	value	val/wt
1	2	12	6
2	1	10	10
3	3	21	7
4	2	15	7.5

this works for fractional knapsack using greedy approach

Consider instance defined by first 'i' items and capacity j ($j \leq W$).

Let $V[i,j]$ be optimal value of such an instance.

$V[i,j]$ = max profit value I can make using first 'i' items and a knapsack of size 'j'.

item 'i'

size of knapsack

$$V(1,j) = \begin{cases} V_1 & j \geq w_1 \\ 0 & \text{otherwise.} \end{cases}$$

final answer will be stored in $V(n, W)$

$V(i, j)$

select i^{th} item → $V(i) + V(i-1, j-w_i)$

reject i^{th} item → $V(i-1, j)$

so, $V(i,j) = \max\left(V(i-1, j),\ V(i) + V(i-1, j-w_i)\right)$

$j \geq w_i$

eg: for above example, if knapsack = 8 kg.

item \ iteration (knapsack wt)	1	2	3	4	5	6	7	8
1	0	12	12	12	12	12	12	12
2	10	12	22	22	22	22	22	22
3	10	12	22	31	33	43	43	43
4	10	15	25	31	37	46	48	58

final answer

if $(V_j == V_i + V(j-w_i))$

values don't change here, $V_{ij} = \max(V_{i-1}, j)$.

$T(n) = O(nW)$

Space $= O(nW)$

↳ backtrack to find what all items I chose → $O(n+W)$

↳ Can we reduce space & time? → Yes.

$V(i, j)$

not required actually.

→ though, it can be done in $O(n+W)$, it is still not a good solution (this problem is

↓

W could be too huge, say 10^9.

→ if profit = very small and size of knapsack = $10^9 = W$
$n = 100$,
then space = $10^9 = W$

↓

this cannot be solved!

algorithm works iff $O(W)$ space is adjustable to RAM and Time $O(nW)$ is fine.

→ W is large, i.e. $\sum v_i = P \leq 10^6$

↓

then I can redefine my space

i, j → now should now represent price,

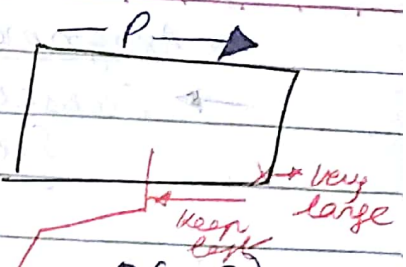
so, $W(i, j)$ is minimum size knapsack required to make a profit of atleast 'j' using first 'i' items

$$W(1, j) = \begin{cases} w & j \leq V_1 \\ \infty & \text{otherwise} \end{cases}$$

i^0 (min)

select → $w_i + W(i-1, j-v_i)$

reject → $W(i-1, j)$



$O(nP)$ time & space

1st time # $\leq W$

I can compute n, P

$W(n, P) \Longrightarrow P = \sum v_i$

given – v_i, knapsack wt W
w_i

max profit I make is of size W.

then,

$V_{i,j}$ = minimum size knapsack such that profit "j" is achieved using first "i" items.

→ backtrack and find items to be picked ?
$O(nP)$ time & space.

not possible with two columns. you can only compute, not backtrack.

if $(P \ll W)$ then

if $(W \ll P)$ then previous solution works fine.

→ if $P = W$ = very large, cannot be solved in $O(n)$ time.

→ Assignment :-

→ Subset sum problem :-

$\{a_0, a_1, \ldots, a_{n-1}\}$

(i) find whether some # is there, so that sum of # = 'x' in the set.

eg: $\{37, 10, 14\}$

find 19 → $\{14, 3, 2\}$ and so on.

(ii) define x_i → 0 (not choose i^{th} value)
→ 1 (choose i^{th} value)

such that $\Sigma x_i w_i = x$.

(iii) if Σa_i's is small,
do something to reduce.

(iv) find largest y, such that $\Sigma a_i x_i = y \leq x$
(no repetition here)

LECTURE - 22

→ Coin exchange continued :-

$O(n\,W)$ → Knapsack size

$O(n\,P)$ → Profit

min (these two), then we implement that part.

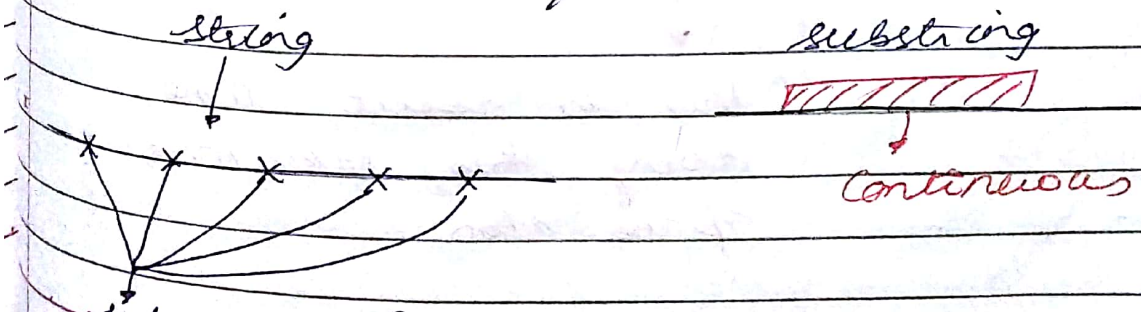
if both are very large enough, we can't search; we can only compute the length

→ LONGEST COMMON SUBSEQUENCE :- (LCS)

↳ LCS is a string 's' obtained by deleting zero or more symbols from original string 'S'.

eg:- president has subsequences → pred, sdn, predent

→ LCS is to find max. length subsequence between two sequence.



subsequence (may not be continuous).

eg:- president & providence.

President
Providence

LCS is "PRIDEN"

so, subsequences — length 'n'

subsequences — m

subsequence — s_m.

→ $O(2^n)$ intersection → exponential solution
hence NPH problem

Dynamic Programming → $O(n^2)$

for string matching :- $O(n^3)$
↓
expected value = $(n \log n)$

→ $O(n)$ → suffix tree
→ $O(n \log n)$ → hashing on substrings.

↓

this is about time, doing this, takes more space also.

$LCS(i,j) \longrightarrow i \quad x_i = y_j$

$\longrightarrow j \quad x_i \neq y_j$

both cannot be there in subsequence

$LCS(i-1,j)$ $\quad L(i,j-1) \longrightarrow j \quad x_m \neq y_m$

so $L(m,n) = \max(L(m,n-1), L(m-1,n))$

else,

$L(m,n) = 1 + L(m-1,n-1)$

Where, $L(i,j) =$ LCS between x_1 and x_i and $y_1 - y_j$

$$\therefore LCS(i,j) = \begin{cases} 1 & x \in \{y_1, y_2, \dots y_j\} \\ 0 & \text{otherwise.} \end{cases}$$

$$LCS(i,1) = \begin{cases} 1 & y_i \in \{x_1, x_2, \dots x_i\} \\ 0 & \text{otherwise.} \end{cases}$$

```
for i = 2 to n
{   for j = 2 to n
    {
        if (x_i == x_j)
            L(i,j) = 1 + L(i-1, j-1)
        else if (L(i-1,j) > L(i, j-1))
            L(i,j) = L(i-1,j)
        else
            L(i,j) = L(i,j-1)
    }
}
```

$O(nm)$

eg: CBABACA and ABCABBA

	C	B	A	B	A	C	A
A	0	0	1	1	1	1	1
B	0	1	1	2	2	2	2
C	1	1	1	2	2	3	3
A	1	1	2	2	3	3	4
B	1	2	2	3	3	3	4
B	1	2	2	3	3	3	4
A	1	2	3	3	4	4	4

answer

$i-1, j-1$	$i-1, j$
$i, j-1$	i, j

so,

Back tracking

$O(n+m)$

```
i=n, j=m
while (i>1 & j>1)
{
    if (x_i == y_i)
      { Print (x_i)
        i=i-1;
        j=j-1
      }
    else if (L(i-1,j) < L(i,j-1))
        j=j-1
    else
        i=i-1
}
```

→ the LCS obtained by backtracking as in above code is NOT a unique sequence. There may be EXPONENTIAL # of sequences

↳ Assignment :- generate a string which has exponential # of subsequences

↳ I'm interested in length only, not the sequence

I need only (current) same column & previous column.

Space reduced to $O(n)$ from $O(nm)$

backtracking is not possible then.

It's variant will be asked in next test.

↳ LONGEST PALINDROME SUBSEQUENCE :-
given a sequence of length 'n', Compute the length of the longest palindrome.

eg: madam, race car

↳ induction on length → easy to solve.
i.e. given length, I can solve it.

1 dimension, 2 dimension?
↳ state?
↳ what to store in table?
↳ Length of longest palindrome?

$T(i,j)$ = longest length palindrome subsequence b/w i & j

$\underset{i}{x} \quad \underset{j}{x}$

of string $x_i, x_{i+1}, \ldots x_j$

final answer at $T(0, n-1)$.

$T(i,i) = 1$.

$$T(i,i+1) = \begin{cases} 2 & \text{if } x_i = x_{i+1} \\ 1 & \text{otherwise.} \end{cases}$$

<u>general case</u>:- if $x_i == x_j$

$L(i,j) = 2 + L(i+1, j-1)$

else if $L(i+1,j) > L(i,j-1)$

$L(i,j) = L(i+1,j)$

else

$L(i,j) = L(i,j-1)$

~~eg~~ eg: ABCABBA and ABCABBA.

	A	B	C	A	B	B	A
A	1	1	1	3	3	3	5
B	X	1	1	1	3	3	4
C	X	X	1	1	1	2	4
A	X	X	X	1	1	2	4
B	X	X	X	X	1	2	2
B	X	X	X	X	X	1	1
A	X	X	X	X	X	X	1

Palindrome LCS:- AB X BA

where X = anyone of {C, BA}

$i-1, j-1$	$X_{i,j}$
$i+1, j-1$	$i+1, j$

d_i

d_{i-1}

d_{i-2}

I need three columns (2 previous diagonals for it).

↳ I need $n \times n$ table and need to fill the upper half only.

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

↳ In case of LCS, changing loops does not matter.

```
i=0 to n
A(i,i) = 1

for l = 1 to n-1
  {for i = 0 to n-1-l
      {  j = i+l.
         if (Xi == Xj)
            T(i,j) = 2+T(i+1,j-1)
         else if T(i+1,j) > T(i,j-1)
            T(i,j) = T(i+1,j)
         else
            T(i,j) = T(i,j-1)
      }
  }
answer at → T(0,n-1)
```

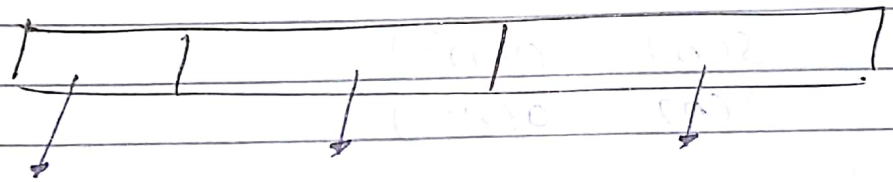
because (0,0) has been taken care of

↳ Backtrack requires linear time.

↳ <u>important problem</u> → because the way we wrote this table is different.

if only length of string is required { 2 previous rows required + current row.

so, $l = 3n$ (for above case)



d_{x-2} d_{x-1} d_x

where d_x = current diagonal

given $r(i,j)$,
I should be able to write code & eq^ns in 'r'.

figure out 3 locations
↓
if overflow
↓
return & start from beginning

↳ but if you dont want this, i.e. need backtracking also, $\begin{cases} T(n) = O(n^2) \\ S(n) = O(n^2) \end{cases}$

↳ <u>Assignment</u> :- try above eg. & write code for it

LECTURE - 23

Longest Palindromic Subsequence :-

S
S^R $\rightarrow$ LPS ✓ (easy to implement)

MATRIX CHAIN MULTIPLICATION :-

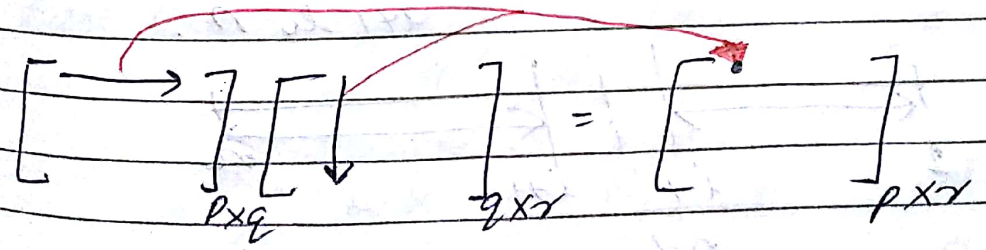
given $A_1, A_2, A_3, A_4, \ldots A_n$.
compute the fastest way i.e. minimum # of multiplications to multiply $A_1 - A_n$.

suppose 'C' $= A_{p \times q} \times B_{q \times r}$

$C_{p \times r} = A_{p \times q} \times B_{q \times r}$

# of computation $= p \times q \times r$

matrix multiplication is ~~commutative~~ not commutative and is associative,
i.e. $AB \neq BA$ but $A(BC) = (AB)C$



q-multiplications & q-1 additions.

so, $A_1(P_0, P_1)$, $A_2(P_1, P_2)$ ..... $A_n(P_{n-1}, P_n)$

$$P_0, P_1, P_2, \ldots\ldots, P_n$$

eg:- different parenthesization will have different # of multiplications for product of multiple matrices.

$A(10, 100)$, $B(100, 5)$, $C(5, 50)$

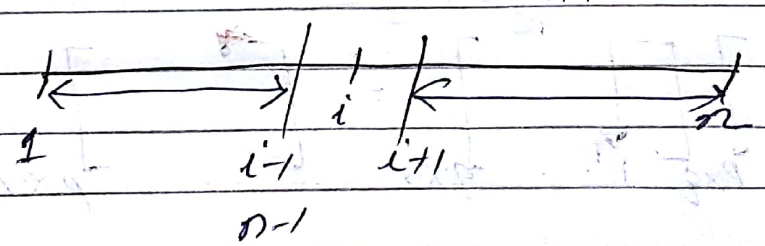
$$(AB)C = 10 \times 100 \times 5 + 5 \times 50 \times 50$$
$$= \underline{7500}$$

$$A(BC) = 10 \times 100 \times 50 + 100 \times 5 \times 50$$
$$= \underline{75000}$$

→ <u>Proof</u>:- Brute force ways to multiply = exponential

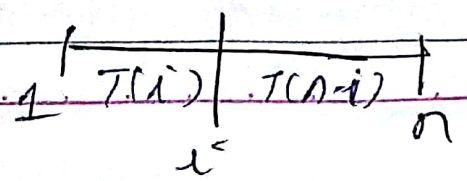
for matrices given,
important is last multiplication I do (i)

previous iterations → I've done matrix
1 to i
i+1 to n.



$$T(n) = \sum_{i=2}^{n-1} T(i) \cdot T(n-i)$$

↓
total # of ways to multiply n-matrices



$$T(n) = \sum_{i=2}^{n-1} T(i) \cdot T(n-i) > T(n-2) + T(n-2)$$

$T(2) = 1$ (# of ways to multiply 2)

$T(3) = 2$

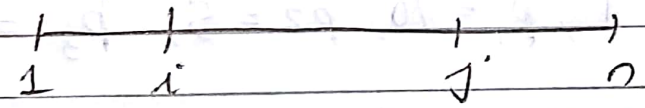
$T(4) = 6$.

So, $T(n) > \sum_{i=2}^{n-1} 2T(n-2) \geq 2^k \cdot T(n-2k) \geq 2^{\frac{n-2}{2}}$

So, $T(n) = \Omega(2^n)$, so brute force does not work here!

So, $\underline{T(100) \geq 2^{49}}$

↳ given matrices from 1 to n,



$M(i,j)$ = minimum # of multiplications required to multiply matrices $A_i, A_{i+1}, \ldots, A_j$

Base case :- $M(i, i+1) \longrightarrow P_{i-1} \times P_i \times P_{i+1}$

$M(i, i) = 0$

i^{th} matrix $\rightarrow P_{i-1} \cdot P_i$ dimensions

$(i+1)^{th}$ matrix $\rightarrow P_i, P_{i+1}$ dimensions.

↳ first I want to compute $M(1,n)$

$\longrightarrow$

$T(i,k)$

Cost = # of computation A_{i-k} + Computation of A_{k+1-j} + # of $A_{i-k} * A_{k+1-j}$

$T(k+1, j)$

where $i \quad k \quad j$ $\quad i \leq k < j$

Best way is :- $T(i,k) + T(k+1, j) + p_{i-1} \cdot p_k \cdot p_j$

Let $M[i,j]$ = min. # of matrix multiplication
then, $M[1,n]$ is answer.

$$M(i,j) = \min\left(M(i,k) + M(k+1,j) + p_{i-1} \times p_k \times p_j\right)$$

eg: $A_1(1 \times 10) \times A_2(10 \times 5) \times A_3(5 \times 2) \times A_4(2 \times 25) \times A_5(25 \times 4)$

$\Rightarrow p_0 = 1, p_1 = 10, p_2 = 5, p_3 = 2, p_4 = 25, p_5 = 4$

$A_1 \times A_2 \times A_3 \times A_4 \times A_5$
$= 1 \times 10 \times 5 \times 2 \times 25 \times 4$

	1	2	3	4	5
1	0	50	60	310	268
2		0	100	600	380
3			0	250	240
4				0	200
5					0

$2_{25} - 4 \rightarrow$

$k = 2, 3$

$k = 2 \rightarrow A_2(A_3 A_4) = 250 + 10 \times 5 \times 25 = 1500$

$k = 3 \rightarrow (A_2 A_3) A_4 = 100 + 10 \times 2 \times 50 = 600$

So, for final answer $A(1,5)$ $(1\times10 \longrightarrow 10\times4)$

$k=1 \longrightarrow 1\times10 \longrightarrow 10\times4$

$A_1 \times (A_2 \text{ — } A_5)$

$= 380 + 1\times10\times4 = \underline{420}$.

$k=2 \longrightarrow (A_1\times A_2)(A_3\times A_4\times A_5)$

$1\times5 \qquad 5\times4$

$= 50 + 240 + 1\times5\times4 = \underline{310}$.

$k=3 \longrightarrow (A_1\times A_2\times A_3)\times(A_4\times A_5)$

$m(1,3) + M(4,5) + 1\times2\times4$

$= 60 + 200 + 8 = \underline{268}$.

answer

$k=4 \longrightarrow (A_1 \text{ — } A_4)\times A_5$

$= \underline{310} + \ldots\ldots$

↳ Have a look at Base case you're considering, whether it is

$0 \text{ ——— } n-1$

or

$1 \text{ ——— } n \longrightarrow$ I am assuming this (default)

↳ Code :-

```
M(i, i) = 0
M(i, i+1) = P_{i-1} x P_i x P_{i+1}

for l = 3 to n
    for i = 1 to n-l
        j = i+l-1
        T[i,j] = T[i+1, j] + P(i-1)xP(i)xP(j)
        for k = i+1 to j
            if (T(i,j) > T(i,k) + T(k+1,j) + P_{i-1}.P_k.P_j)
                T(i,j) = T(i,k) + T(k+1,j) +
                        P_{i-1}.P_k.P_j
```

$T(n) = O(n^3)$
$S(n) = O(n^2)$ ⟶ I fill lay the matrix
$O(n^2) \times$ lookup (1)
$T(n) = O(n^3)$

⟶ Backtracking :- 268 (answer)

how this min # of computation was reached?

then computation I do is when I do backtracking?
$O(n^2) \longrightarrow T(n)$

⟶ we also store the value of 'k' → called predecessor f^n (π)

$$M(i,j) = \min_{i \le k \le j} \left(\underset{i}{\text{———}} \overset{k}{\times} \underset{j}{\text{———}} \right)$$

then

$$\underset{\text{predecessor}}{\pi(i,j)} = \arg\min_{i \le k < j} (\quad)$$

$$\min = \min_{1 \le i \le n} a_i$$

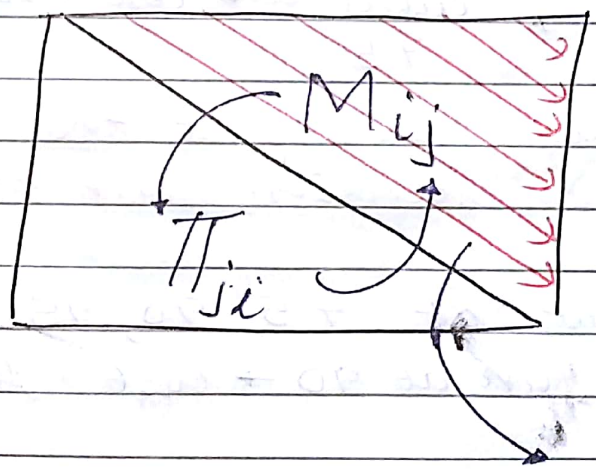
$$k = \arg\min_{1 \le i \le n} a_i$$

so, $a[k]$ is minimum.

↳ So, If I maintain the predecessor also, then I can find the multiplication order in $O(n)$

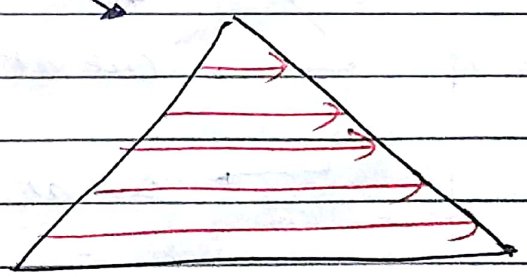
```
MM(i, j)
{ if (j > i)
  {
      k = π(i, j)
      MM(i, k)
      MM(k+1, j)
      print k
  }
}
```

↳ But, I'm not using the lower half of the matrix, I can use it as π



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

↳ I can compute all this in $O(n^3)$ time and $O(n^2)$ space.



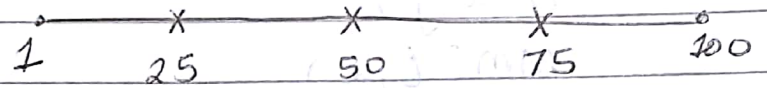
↳ this cannot be optimized.

LECTURE - 24

↳ **Rod cutting Problem :-**

given a rod of length 'n', find the way in which cutting cost is minimum

eg: $n = 100$, cut at 25, 50, 75



Cut at 25 → cost = 100 + 75 + 50 = 225

cut at 50 → cost = 100 + 50 + 50 = 200 (minimum)

cut at 75 → cost = 100 + 75 + 50 = 225

In this case, cutting at the centre gives optimum solution, but that's not always true

eg:- cut at 75, 90, 95, $n = 100$

cut at 90 → cost = 100 + 90 + 10 = 200

cut at 75 → 100 + 25 + 10 = 135 (optimum)

cut at 95 → 100 + 95 + 90 = 285

↳ So, we cannot generalize it, to which side we need to start with.

→ Consider a generic problem,
you've been given n-indices where you need to cut the rod in (n+1) pieces.
then, $l(i)$ = location I need to cut
$l(0) = 1$ (left end point)
$l(n+1) = 100$ / length of the rod.

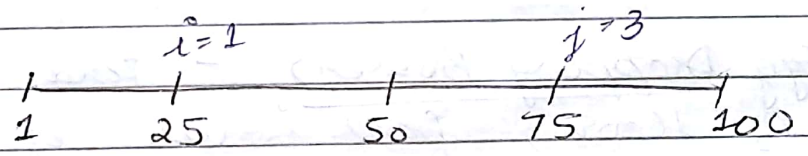
so,

$l(i)$	1	25	50	75	100
i	0	1	2	3	4

Induction is on the # of cuts in the graph.

$T(i, j)$ = cost of cutting from 'i' to 'j'
$0 \le i \le j \le n+1$.
# of cuts = $j - i - 1$.

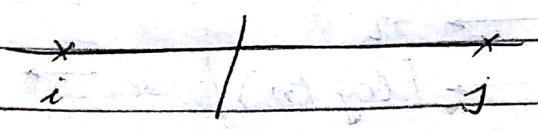
eg:- $i = 1, j = 3$



# of cuts = $j - i - 1$.

so, $T(i, i+1) = 0$ (no cuts required)
$T(i, i+2) = l(i+2) - l(i)$
I need only one cut at location $i+1$.

$$T(i, j) = l(j) - l(i) + \min_{i<k<j} \{T(i,k) + T(k,j)\}$$



k → is a random location b/w i & j

and $T(0, n+1)$ = final answer

$T(n) = O(n^3)$ → visit n^2 states n-times

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

(fill diagonally)

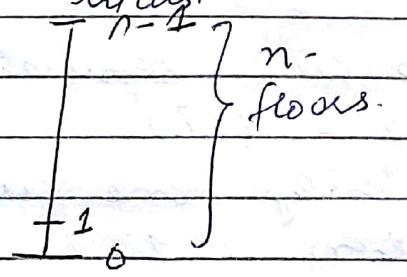
so, $h = 2$ to n
$i = 0$
$j = h$ to n
if $(j - i > 2)$
$k = i+1$ to j and find min k
else
$L(i,j) = A(j) - A(i)$

→ Backtracking :- find where we did the change
→ $\pi(i,j)$ → store the value of 'k', it shows the value of parent selected.

Instead of a new 2D matrix, use $L(j,i)$ to store values for backtracking.

$O(n)$

→ Egg Dropping Problem :- Find the lowest # of floor, from which if an egg is dropped, it breaks in minimum # of trials.



K eggs are similar.
drop(egg)
→ breaks
→ does not break.

# of trials

$(n, 1) \longrightarrow n$

$(n, \infty) \longrightarrow \lceil \log(n) \rceil$ → i.e. I don't need more than 'n' eggs in any case.

trials with eggs

floors

eggs

$\lceil \log_2 n \rceil \leq \min \leq n$

$\log n < \min \leq n$

since, we do not need more than $\log_2 n$ eggs, then

$1 < K < \log n$

eg:- $n = 30$,
$K = 1 \longrightarrow$ 'n'.
$K = 2 \longrightarrow 8$

drop at 8 → doesn't break drop
break check(1-7)

1	8	15	21	26	30

$8 + (8-1) = 15$.
repeat

so, generic case →

$i \quad 2i-1 \quad 3i-2 \quad 4i-3 \quad \geq n$

$x \quad x \quad x \quad x$

$x-1 \quad x-2 \quad x-3$.

$1 + 2 + \cdots + i \geq n$

$\frac{i(i+1)}{2} \geq n$ ← considering max 'i' trials

↳ smallest 'i' such that above holds true.

eg: $n = 15$.
we need 5 trials.

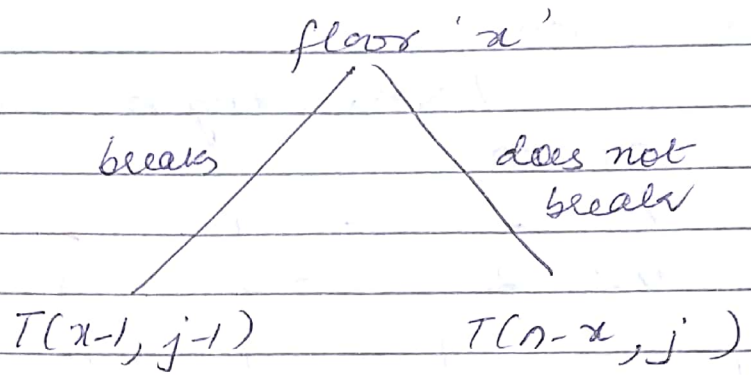
so, $n = 30 \longrightarrow$ eggs $\longrightarrow$ trials required

30	1	30
30	2	8
30	3	6 (Binary + 2 egg part)
30	4	6 (Binary + Binary + 2 egg part)
30	5	5

given, $T(i, j)$ $\longrightarrow$ $T(n, k)$ needed.

$T(i, 1) = i \;\forall i$

i → # of floors, j → # of eggs



$$T(i,j) = \min_{0 < x < i} \left[\max \{ T(x-1, j-1), T(n-x, j) \} \right] + 1$$

$T(n, k) = (\text{final cost})$.

$\Pi(i, j) = K$ $\longrightarrow$ where you drop the first egg

$K \longrightarrow O(n)$

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

$S(n) = O(nk)$

→ Homework

B = Boolean variable in length 'n'

0 0 0 0 ... 0 1 1 - - - 1

find the point where '0' switches to '1'.
Check that there is no '1' b/w 0's and no '0' b/w 1's.

→ Longest Increasing Subsequence (LIS) :-

sort the sequence and apply LCS

→ $T(n) = O(n^2)$

→ $S(n) = O(n^2)$

$LIS(i)$ = length of LIS ending at a_i; including (a_i)
$LIS(0) = 1$.
and $LIS(n-1) = n$, length of longest sequence which includes last element.

$LIS = \max_{0 \le i \le i-1} LIS(i)$ → not necessarily includes $LIS(n-1)$, $a(n-1)$ as last term. I don't actually know, what is the last term.

$LIS(i) = 1 + \max_{0 \le j < i} LIS(j)$
such that
$a_j \le a_i$ → to keep a track of increasing sequence

$i = 1$ to n $(i<n)$ smallest value = 1
$j = 0$ to i $(j<i)$ becz, smallest LIS = 1 (in length)
if $a_j \le a_i$ && $LIS(j)+1 > LIS(i)$
$LIS(i) = LIS(j)+1$
$\pi(i) = j$

Keep a track of π value, then backtracking is done in $O(n)$;
so, it can be solved in $O(n \log n)$. (sort & solve)

$T(n) = O(n^2)$
$S(n) = O(n)$ → but what if $n = 10^6$?

so, $\forall i$, what is the first location where I get a # smaller than this one?
→ Using stack → $O(n)$

LIS	1	2	1	2
values	2	3	1	10

but max LIS = 3 (2, 3, 10).

eg:- Find the longest sequence increasing (when number comes, you have an option to add it to right or left of smallest).

12,	7,	10,	8,	1,	6,	5,	14,	4	9	10
0	1	0	1	0	1	1	0	1	1	1

What ever is in between 2 # has to be discarded, thus what # to include in the sequence has to be carefully chosen.

actual → 2, 4, 5, 6, 7, 8, 9, 10 → LIS = 8

1, 7, 12, 14 (length 4)
10 & 8 are discarded becz anything b/w 1 & 14 will be disturb

$LIS(i) =$ a_i (i) — (n-1)

$LDS(i)$ a_i (i) — (n-1)
↑ included

$$\max_{0 \le i \le n-1} LIS(i) + LDS(i) - 1$$

(∀i)

$$\boxed{\max_{\forall i} LIS(i) + LDS(i) - 1}$$

$\forall i \begin{cases} LIS(i) = 1 , & \pi_1(i) = n \\ LDS(i) = 1 , & \pi_2(i) = n \end{cases}$

for $i = n-1$ to 0
  for $j = i+1$ to n
    if $a_j \geq a_i$ && $LIS(j)+1 > LIS(i)$
      $LIS(i) = LIS(j)+1$
      $\pi_1(i) = j$
    if $a_j \leq a_i$ && $LDS(j)+1 > LDS(i)$
      $LDS(i) = LDS(j)+1$
      $\pi_2(j) = j$

π_1 and π_2 gives elements in LIS & LDS.

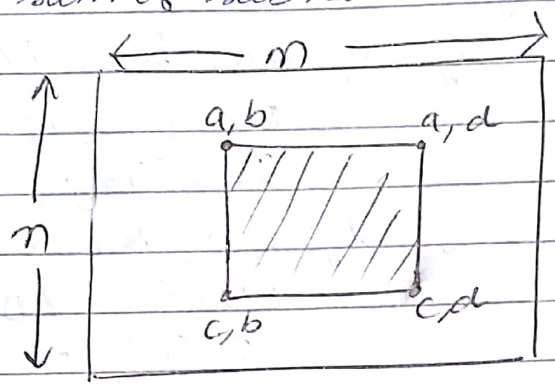
$T(n) = \Theta(n^2)$
$S(n) = \Theta(n) \longrightarrow$ 4 arrays of length 'n'.

where LIS = LIS till 'i' from '0', a_i included
LDS = LDS from i to n-1 including a_i

LECTURE - 25

MAXIMUM SUBMATRIX PROBLEM:-

given a matrix 'm' having +ve & -ve nos.
find a submatrix so that the sum of submatrix is max.

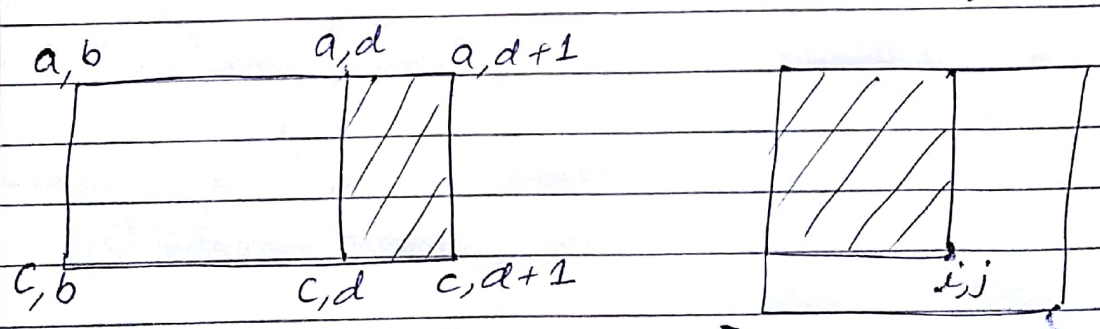


initial maxsum = -∞

```
for a = 0 to n
    for b = 0 to m
        for c = a to n
            for d = b to m
                for i = a to c+1
                    for j = b to d+1
                        sum = sum + A[i,j]
                        if (sum > max)
                            sum = max
```

iterating over all possible matrices.

$T(n) = O(n^3 m^3)$



possible cases.

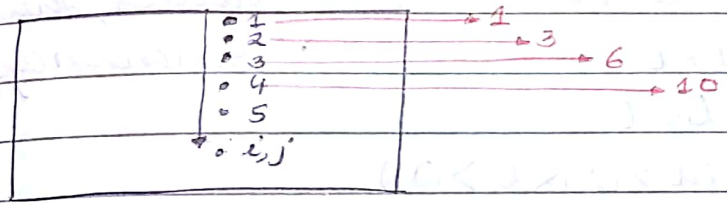
value of submatrix where one end point is (0,0) and other is (i,j)

$$sum = M[c,d] - M[a-1,d] + M[c,b-1] + M[a-1,b-1]$$

$$\underbrace{O(m^2n^2)}_{} < O(n^3m^3)$$

can replace inner two loops
precomputations = $O(mn)$

↳ create a matrix that stores column sum



$M(i,j)$ represent sum of all elements above it in the same column.

```
i = 0
for j = 1 to m                    } O(m)
    M[i,j] = A[i,j]
for i = 1 to n
    for j = 0 to m                } O(mn)
        M[i,j] = M[i-1,j] + A[i,j]
```

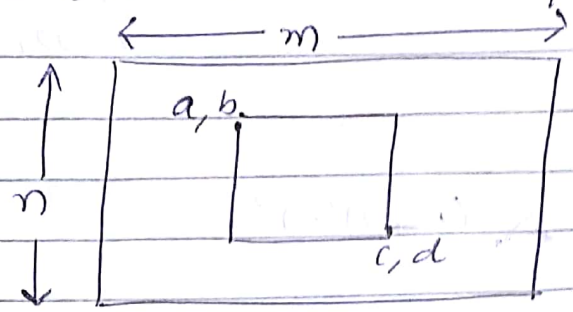
```
for i = 0 to n
    for j = 1 to m                } O(nm)
        M[i,j] = M[i,j-1] + M[i,j]
```

pre-computation

so, $T(n) = O(n^2m^2)$

so, given 'A' we can compute M in $O(nm)$

→ now I want a square matrix



$a \rightarrow 0$ to n
$b \rightarrow 0$ to m
↓
c and d don't have abitary values, there is a dependency.

```
for a = 0 to n
  { for b = 0 to m
      { for l = 0 to n
          {  c = a + l
             d = b + l
             if (a > 0 & b > 0)
                 sum = m(c, d) - m(a-1, d) - m(c, b-1) + m(a-1, b-1)
             else if (a > 0)
                 sum = m[c, d] - m[c, b-1]
             else if (b > 0)
                 sum = m[c, d] - m[a-1, d]
             else
                 sum = m[c, d]
          }
      }
  }
```

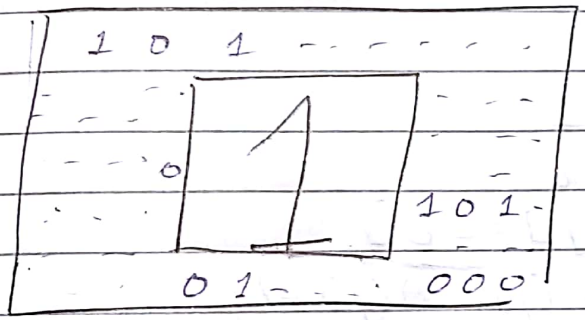
→ $T(n) = O(n^2 m)$
given $n < m$.
otherwise
$T(n) = O(m^2 n)$

So, rectangular → $O(n^2 m^2)$
square → $O(n^2 m)$
fixed size square → $O(nm)$
highest sum subarray → $O(n)$

→ given an $n \times m$ matrix,
we need a $k \times k$ matrix

$$\left.\begin{array}{l} a = 0 \text{ to } n-k \\ \quad b = 0 \text{ to } m-k \\ \quad\quad c = a+k \\ \quad\quad d = b+k \\ \quad\quad \vdots \end{array}\right\} O(nm)$$

eg: Binary matrix :-
find a rectangular matrix where everything is '1' & of max. size.



eg seen = # of elements = $(c-a)\cdot(d-b)$
then all entries are one.

$O(n^2m^2)$ algorithm → I've only 0 & 1's so I should be able to answer it faster

↓

consider the last row, take sum of all 1's in the column & then the array we get, represents it with histogram soln

$$O(m) \times O(n) = \underline{O(nm)}$$

Sol^n:

```
i = n-1
    for j = 0 to m
    {       A(j) = m(i,j)
            k = i ~~and k~~
            while ( k>0 && m(k,j) == 1)
            {       k--;
                    A(j) = A(j) + m(k,j)
            }
    }
    max = findBestSol^n (A, 0, m-1)
```

```
i = n-2 to -1
    j = 0 to m
        if (A[j] == 0)
                A(j) = A(j) -1;
        else {
                if (m(i,j) == 0)
                        A(j) = 0
                k = i
                while (k>0 && m[k,j] == 1)
                { A(j) = A(j) + 1
                  k = k-1
                }
        }

sol^n = findBestSol^n( A, 0, m-1) ---> O(m)
if (sol > max)
        max = sol
```

↳ The claim is that total time I spend in updating 'vector A'

$A \rightarrow O(nm)$.

we need a square sub matrix such that every entry is 1 in rectangle of $n \times m$

$mes = min(h, l)$

$h = min(r(i), l(i))$

$l = r(i) - l(i) + 1$

this gives a square matrix.

the way we do 'a' & 'b' etc. is still the same.

→ I also want to find $k \times k$ where all entries are 1 → trivial.

If return value is $> k^2$
return yes
else return no.

similarly, all zeroes:

LECTURE - 26

V = Vertex, E = Edges $\rightarrow$ Graph $G = (V, E)$

then for a graph,

$$E \leq V \times V = \{(a,b) \mid a, b \in V\}$$

E → # denoted by m

V → # denoted by 'n'

$\Rightarrow \underline{m \leq n^2}$. (directed)

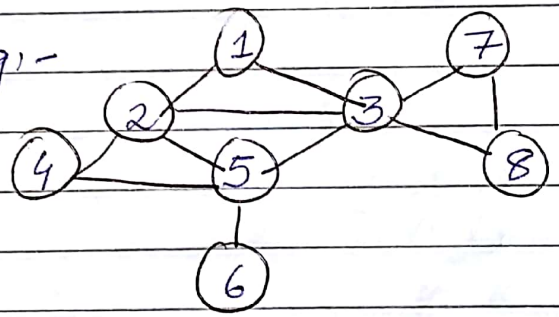
↳ In a simple graph, self loops are not allowed,

so, $m \leq n^2 - n$ (directed)

↳ If the graph is undirected, then

$$m \leq \frac{n^2 - n}{2}$$

eg:-



$V = \{1, 2, 3, 4, 5, 6, 7, 8\}$

$E = \{(1,2), (1,3), \ldots\}$

$n = 8$

$m = 18$

↳ Types of Graphs :-

(i) Edge types :-
- → undirected (unordered pair)
- → directed (ordered pair) i.e. $(a,b) \neq (b,a)$
- → loops : eg: (a,a)

(ii) weighted graphs :- edges have some weight associated with them

Representation :-

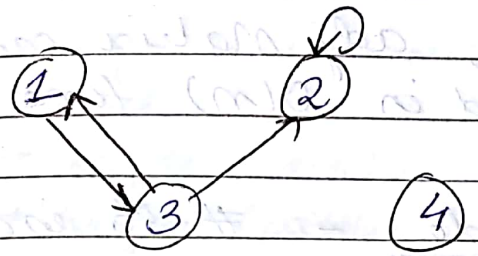
1. **adjacency matrix** :- (used in case of (boolean) directed graph)

→ space constraint

↓

in a sparse graph
$m \leq n^2$
but can be small enough

eg:



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

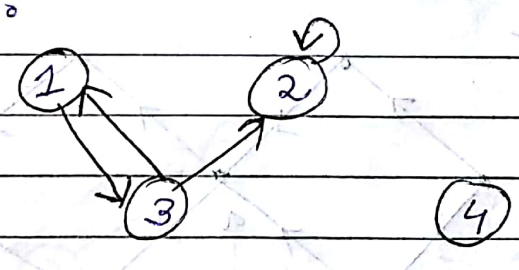
$S(n) = O(n^2)$

↓

bits in boolean, bytes otherwise

2. **Adjacency list** :-

eg:



Vertex

1 → 3 → NULL
2 → 2 → NULL
3 → 1 → 2 → NULL
4 → NULL

$S(n) = O(n+2m)$
$= O(n+m)$

↓

each edge represented twice.

3. Edge list :- list all edges in any order.
$S(n) = O(m)$.

Imp
↳ Adj Matrix ⟶ Adj list ~~+++++~~ ~~Adj~~ ~~list~~
given Adj Matrix,
Adj list can be produced in $O(n^2)$ time.

Imp
↳ given edge list, adj matrix can be produced in $O(n)$ time.

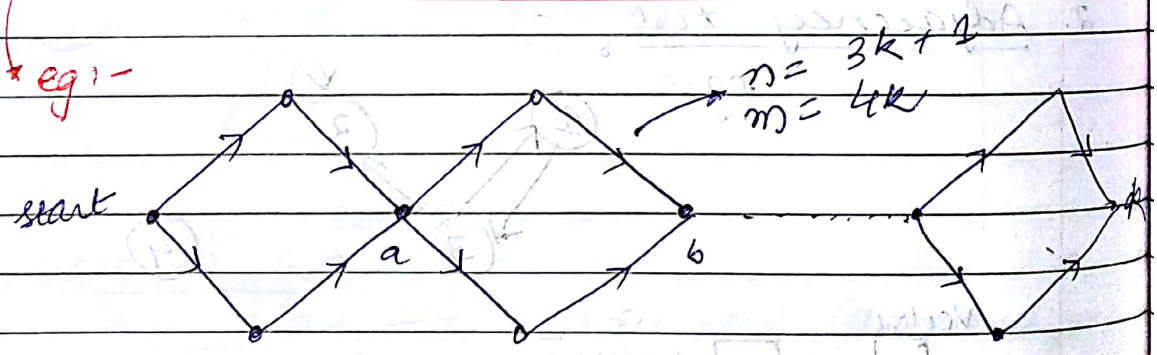
↳ degree of a node :- # of incident vertices.

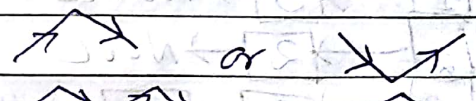
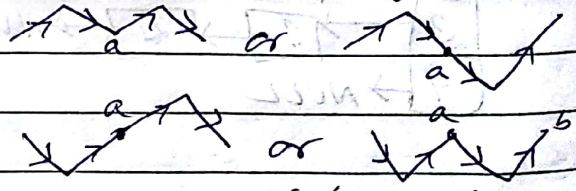
Σ degree $= 2n$

NOTE :- don't write consider "all paths"...
↓
NPH
exponential time

If "consider all paths" found in exam,
ZERO

eg :-



path to a ⟶ 
path to b ⟶ 

So, for above graph, it's 2^k "all paths" possible for a path distance 'k'.

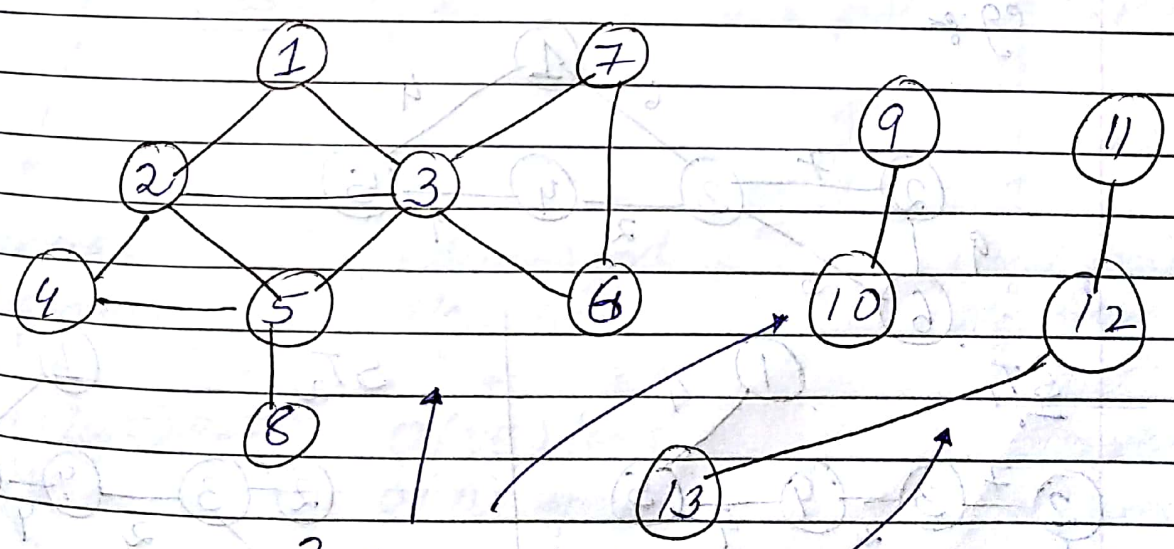
↳ <u>Path in a graph</u>:- a path in an undirected graph $G = (V, E)$ is a sequence 'P' of nodes $v_1, v_2, \ldots v_k$ with a property that each consecutive pair v_i, v_{i+1} is joined by an edge E.

↳ <u>walk</u>:- path in which edges / nodes may repeat

↳ <u>path</u>:- generally means no repetition of nodes.
- → Length (path) = # of edges in the graph
- → Weight (path) = wt of edges in the path

↳ <u>connected graph</u>:-
- ↳ two vertices are connected if there is a path b/w them.
- ↳ an undirected graph is connected if every pair of nodes u & v there is a path b/w u & v.
- ↳ a connected component of G is a <u>maximal</u> connected subgraph G.

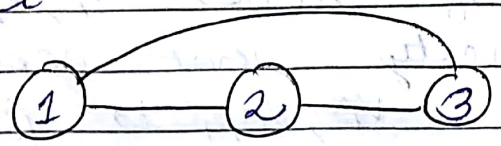
eg:-



3 connected components

↳ **Cycle** :- is a path in which $v_i = v_k$, where $k - i > 2$ and the first $(k-i)-1$ nodes are distinct.

eg:-

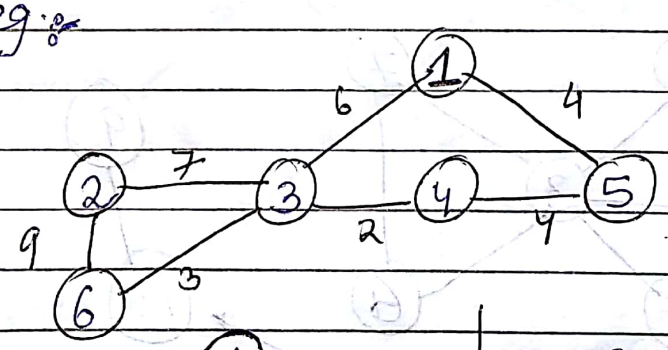


↳ **Tree** :- is an (undirected) graph that is connected and does not contain a cycle
↳ a union of two or more trees is called a forest.

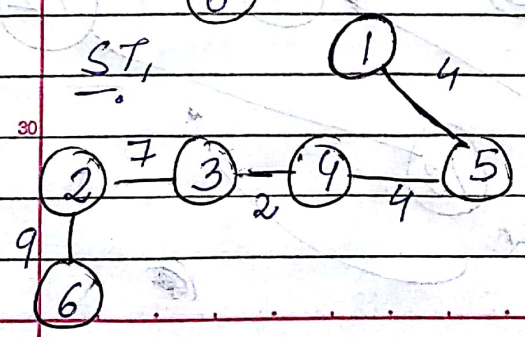
Theorem :-
Let G be an undirected graph on n nodes. Any two of the following statements imply the third :-
↳ G is connected
↳ G does not contain a cycle.
↳ G has n-1 edges.

↳ **Spanning Tree** :-
a spanning tree 'T' of Graph 'G' is a subgraph of G, which is a tree with "all" 'n' nodes.

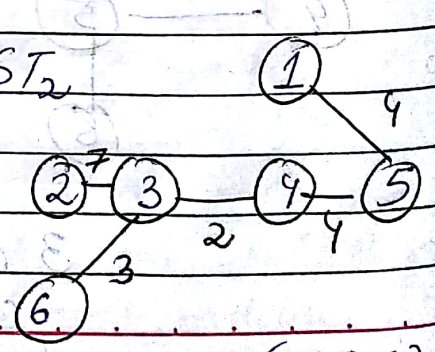
eg:-



ST_1



ST_2



≡ (MST)

may/maynot be unique

MINIMUM SPANNING TREE (Min ST)

a spanning tree with minimum weight

- Prim's algorithm
  - $O(V+E)$
  - minheap, PQ.
- Kruskal's algorithm
  - $O(E \log E)$
  - array of edges to sort.

may/maynot be unique

maximum Spanning Tree :-

- a spanning tree with max. weight.
- first do reverse of what done in minimum spanning tree, i.e. select edge with highest weight, you'll get minimum spanning tree.

So, it is solvable, it is a 'P' class problem.

Shortest Path Problem :-

- Single ~~Path~~ Source Shortest Path (SSSP)
  - Dijkstra's algorithm
    - $O((E+V)\log V)$ time
    - $O(n)$ space
  - Bellman Ford Algorithm
    - $O(VE)$ time
    - $O(n)$ space
- All pair Shortest path (APSP)
  - Floyd warshall Algorithm
    - $O(n^3)$ time
    - $O(n^2)$ space.

edge list

but, Longest Path Problem is NPH.

↓

It reduces to Hamiltonian path/cycle problem.

so, just as max. spanning tree can be found easily using min. spanning tree algorithm,

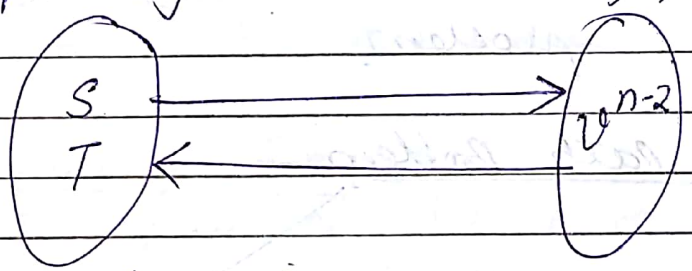
we cannot find longest path using shortest path algorithm.

→ Cayley's theorem :-

In, Complete graph 'G' with n-nodes,

# of spanning trees possible = n^{n-2}.

Spanning Tree $V = \{1, 2, \dots n\}$



$V^2 = \{(a, b) \mid a, b \in V\}$ ordered pair

$V^3 = V^2 . V$

above is Bijection function (one to one - onto)

so, V^{n-2} ⇒ needs a vector of length $(n-2)$.

or

Cardinality $= (n-2)$

$|V| = n$

↳ is cardinality

so, $|V^{n-2}| = n^{n-2}$.

with x - vertices nodes.

0	not possible
1	$n-1$ $\left[\lfloor n-1 / \lfloor n-2\right]$
2	$(n-1)(n-2)$ $\left[\lfloor n-1 / \lfloor n-3\right]$
⋮	⋮
n	$\lfloor n-1$ $\left[\lfloor n-1 / \lfloor n-(n-1)\right]$

adding for total paths that exists

$$\lfloor n-1 \sum_{i=1}^{n-2} \frac{1}{\lfloor i} \longleftarrow$$ # of paths exist.

if n- vertices, (n is not fixed) then

$$\lfloor n \sum_{i=1}^{n-2} \frac{1}{\lfloor i}$$ paths.

so, writing "Consider all paths" = "0" marks.

LECTURE - 27

↳ Transitive closure of a graph

$T(i,j) = 1$ iff there is a path from i to j in directed graph.

We've been given adjacency matrix

$$A_{ij} = \begin{cases} 1 & (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

↳ If I apply BFS or DFS at i, it gives me all vertices which I can reach.

$$i \xrightarrow{\text{DFS/BFS}} j \qquad O(V+E)$$

If we repeat this on every vertex, we get an $O(VE)$ algorithm.

↳ Thinking it as DP problem,

$T^{k}(i,j) = 1$ iff $\exists$ a path from i to j in which only the first 'k' vertices are allowed as intermediate nodes.

→ assumption :- vertices # from 1 to n

induction :- k

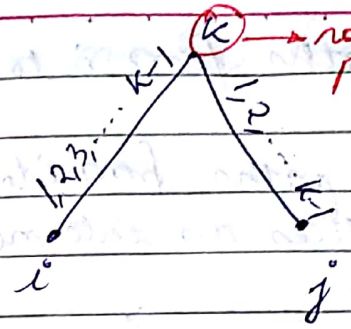
$$i \xrightarrow{1, 2, \ldots, k} j$$

$T^{0}(i,j)$ = no intermediate vertex, only directed edge

↳ given $T^{k}(i,j) \xrightarrow[\text{compute}]{\text{how to}} T^{k+1}(i,j)$

or

$$T^{k-1} \longrightarrow T^{k}$$



not captured in previous iteration.

$T(k-1)$ $1, 2, \ldots (k-1)$

$T(k)$ $1, 2, \ldots k$

$$T^k(i,j) = \begin{cases} 1 & \text{if } T^{k-1}(i,j) = 1 \\ & \text{or} \\ & T^{k-1}(i,k) = 1 \text{ and } T^{k-1}(k,j) = 1 \\ 0 & \text{otherwise} \end{cases}$$

→ Warshall's Algorithm (for transitive closure)

```
for i = 0 to n
    for j = 0 to n
        T(i,j) = A(i,j)
```

initialization.

T^0
T^1
T^2
⋮

need $O(n^2)$ space if keep them all.

```
for k = 0 to n
    for i = 0 to n
        for j = 0 to n
            if T(i,k) == 1 & T(k,j) == 1
                T(i,j) = 1
```

update T^{k-1} to T^k.

$O(n^3)$.

we need to compute n-matrices here, and computing one matrix = $O(n^2)$

$N(i,j)$ = # of paths from i to j
and
$N^k(i,j)$ = # of paths from i to j with exactly k vertices as intermediate vertices.

$$N(i,j) = N(i,j) + N(i,k) * N(k,j)$$

# of paths from i to j can be exponential

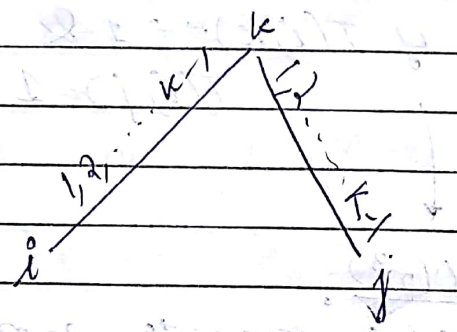
→ <u>All pair shortest path (Floyd Washall)</u>.

Given a weighted graph find shortest path b/w any two nodes.

$D^k(i,j)$ = Shortest path from $i \longrightarrow j$ using first 'k' intermediate vertices.

$D^0(i,j)$ = given (wt of an edge from i to j) (direct edge)

$$D^{k-1}(i,j) \longrightarrow D^k(i,j) \longrightarrow \underbrace{D^n(i,j)}_{\text{shortest path}}$$



$$D^{k-1}(i,k) + D^{k-1}(k,j) \geq D^k(i,j)$$

So $D^k(i,j) = \underbrace{\min}_{\text{shortest path}} \{ D^{k-1}(i,j) , D^{k-1}(i,k) + D^{k-1}(k,j) \}$

```
for K = 1 to n
    for i = 1 to n
        for j = 1 to n
            if D^{k-1}(i,j) > D^{k-1}(i,k) + D^{k-1}(k,j)
                D^k(i,j) = D^{k-1}(i,k) + D^{k-1}(k,j)
            else D^k(i,j) = D^{k-1}(i,j)
```

$O(n^3)$ time

It takes $O(n^3)$ space, so I allow only one matrix & I update it.

→ floyd warshall's algorithm :-

```
for i = 0 to n
    for j = 0 to n
        D(i,j) = A(i,j)
```

$T(n) = O(n^3)$

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

```
for k = 0 to n
    for i = 0 to n
        for j = 0 to n
            if (D(i,k) + D(k,j)) < D(i,j)
                D(i,j) = D(i,k) + D(k,j)
                π(i,j) = k
```

Order of loops here does not matter in inner loops of i & j.
'k' should be outer loop only, so order is important.

multiplying two matrices :-

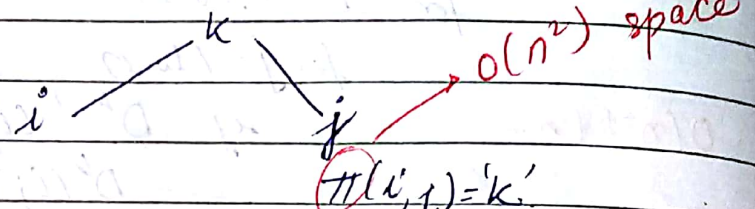
```
i = 0 to n
    j = 0 to n
        k = 0 to n
            C(i,j) = C(i,j) + A(i,k) * B(k,j)
```

exactly same as floyd warshall, order of loops here doesn't matter.

A(ij) gives only the shortest path, but I want, what gives me the shortest path.

↓

I use Π matrix here.



$\Pi(i,j) = 'k'$

↳ print paths.
$\{ i \to \pi(i,j)$
$\pi(i,j) \to j$

initialization :-

for $i = 0$ to n
  for $j = 0$ to n
    $\Pi[i][j] = NIL / i$

↓ to avoid NULL ptr.

$O(n^2)$ space } floyd
$O(n^3)$ time } warshall.

↳ # of shortest paths from i to j $= N(i,j)$

$N(i,j) = N(i,k) * N(k,j)$

↓

if path I discovered is of same wt as other shortest path, # of shortest path increase by 1.

$N(i,j) = N(i,j) + 1$.

Single source shortest paths :- given a weighted graph & a vertex 's', find shortest paths from 's' (→ source) to all vertices.

↳ floyd warshall $(O(n^3))$

$O(n^2)$ space is a problem

↓

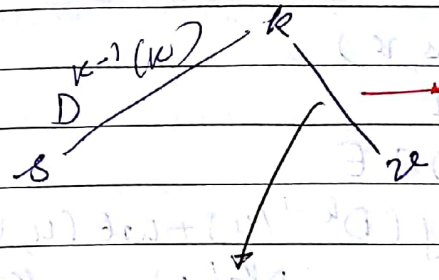
If $n = 10^4$?

$O(VE)$ algo using $O(V)$ space is what I use.

$s \xrightarrow{1, 2, \ldots k} v$

$D^0(v)$ = direct edge

I need $D^n(v)$



→ I'm not storing the path from k to v.

$D^k(v)$ tells from only $s \to v$. (shortest path)

I want to force it to be an edge, then I know the wt of edge.

$s \sim\sim v$

$D^k(v)$ = shortest path from s to v with length atmost 'k'.

$D^1(v) = wt(s, v)$

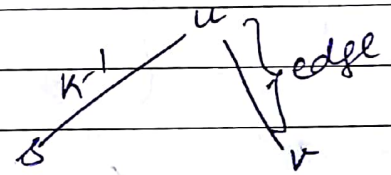
$\vdots$

$D^{k-1}(v)$

$D^k(v)$ → k-edges in path.

$\vdots$

$D^{n-1}(v)$



k = 2 to n-1

for all vertices $u, v \in V$

$(u, v) \in$ Edges

if $D^{k-1}(u) + wt(u, v) \leq D^{k-1}(v)$

$D^k(u) = D^{k-1}(u) + wt(u, v)$

Can I merge these two loops?

It's a repeat for each vertex, bec every edges is used only once. (relaxation).

use ∀edge do ....

$D^k(v)$ = length of shortest path from u to v of length atmost 'k'.

$D^1(v) = wt(s, v)$
$k = 2$ to $n-1$
$\forall (u,v) \in E$
if $(D^{k-1}(u) + wt(u,v) < D^{k-1}(v))$
$D^k(v) = D^{k-1}(u) + W(u,v)$
else if $(D^k(v) > D^{k-1}(v))$
$D^k(v) = D^{k-1}(v)$

↳ <u>Bellman Ford</u> :-

initialization
for v in V
$D[v] = wt(s, v)$

$\pi(u) = s$
$N(v) = N(v) + N(u)$

this ensures L=1 is taken care of.

relaxation
for k = 2 to n
for all edges (u,v)
if $(D(u) + wt(u,v) < D(v))$
$D(v) = D(u) + wt(u,v)$
$\pi(v) = u$

$D(v) \leq D^k(v)$ can be ensured.

$O(n)$ space
$O(VE)$ time

print path(v)
if $(\pi(v) \neq s)$
print $(\pi(v))$

$N(v)$ = # of shortest path from u & v.

shortest path $s \longrightarrow v$ of length exactly 'l'

I need to control # of intermediate nodes → $l-1$ intermediate nodes.

length of path

$D^1(v) = wt(s, v)$

$k = 2$ to $l+1$ $\forall$ edges (u,v)

if $D^k(v) > D^{k-1}(u) + wt(u,v)$

$D^k(v) = D^{k-1}(u) + wt(u,v)$

if $D(v) > D_1(u) + w(u,v)$

$D(v) = D_1(u) + w(u,v)$

adding exactly one vertex/edge in each iteration

↳ find shortest path from s to v of length

↳ exactly 'l'

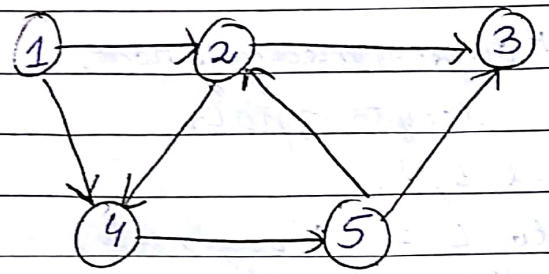
↳ atleast 'l'

↳ atmost l

one of them is test/exam.

LECTURE-28

eg:



↳ Initially give unvisited nodes.

Data structure

unvisited nodes goes in here

```
u = Delete(DS)
    if (u == not visited)
        Add v.
    else if (v == visited)
    {
        :
    }
```

↳ True - false checks only two things, I need to check three
- → unvisited (red) (R)
- → in queue (blue) (B)
- → visited (green) (G)

↳ If graph is having different connected components, apply algo above repeatedly.

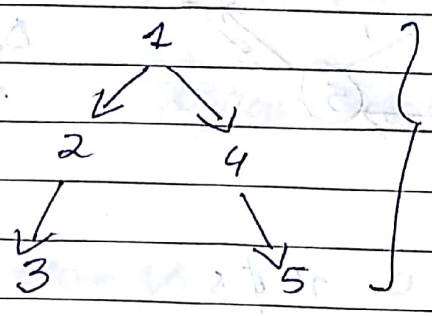
```
visit(s):
    enqueue(s), color(s) = B
    while(Q)                                  add = O(1)
        u = Dequeue , color(u) = G            delete = O(1)
        ∀(u,v) ∈ E
            if (color(v) == R)
                Enqueue(v).
                color(v) = B
                π(v) = u
```

$\pi \rightarrow$

$\emptyset$	1	1	2	4
~~1~~	~~2~~	~~4~~	~~3~~	~~5~~

$\longrightarrow (\pi(v), v) \in E$ $\longrightarrow$ BFS traversal.

gives n-1 edges



BFS tree

(edge considered only once)

$O(V+E)$ time complexity

$\hookrightarrow$ I replace queue by stack. (DFS).

$\forall v \in E$, $C(v) = Red$
push(s)
$C(s) = Blue$
while (S)
u = pop
$C(u) = Green$.
$\forall (u,v) \in E$
if $(C(v) == Green)$
push(v)
$C(v) = Blue$
$\pi(v) = u$.

if $(C(u) = Green)$
do nothing
else

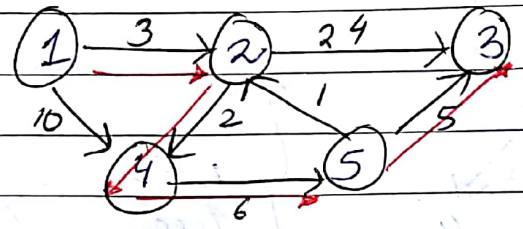
not required.

3	5
~~2~~	5
~~5~~	~~4~~
~~4~~	1
~~2~~	1
~~1~~	

vertex can go to stack more than once, but edge considered (deleted) only once

$O(V+E)$ time complexity

→ now, consider a weighted graph



DS = min heap
$w : E \rightarrow R^+$

$p(s) = 0$
$p(v) = \infty$, $v \notin s$
$\pi(s) = s$
$color(v) = red$
Enqueue (s) ⟶ <u>dijkstra's algorithm</u>
color(s) = Blue
while (Q)
u = delete_min (Q)
↓
$\forall (u,v) \in E$

if (color(v) == R)
$p(v) = p(u) + wt(u,v)$
enqueue (v)
$\pi(v) = u$
color(v) = Blue

else if (c(v) == Blue)
& $p(v) > p(u) + wt(u,v)$
$p(v) = p(u) + wt(u,v)$
$\pi(v) = u$

add = $O(v)$
delete = $O(v)$
decrease key = $O(v)$

<u>Binary Heap</u> :-

add	: $O(\log V)$	$O(v+E)\log V$
delete	: $O(\log V)$	
decrease	: $O(\log V)$	

<u>Fibonacci Heap</u> :- $O(V \log V + V)$ → V additions
$+ E \log V)$ → delete key V times.

↳ what is the above algorithm doing?

$p(v) \longrightarrow$ wt of the shortest path from 's' to 'v'.

↓

dijkstra's algorithm

$p(1) = 0$

$p(4) = 0 + 10 = \cancel{16}$ $\quad 3 + 2 = 5$

$p(2) = 0 + 3 = 3$

$p(3) = 3 + 24 = \cancel{27}$ $\quad 11 + 5 = 16$

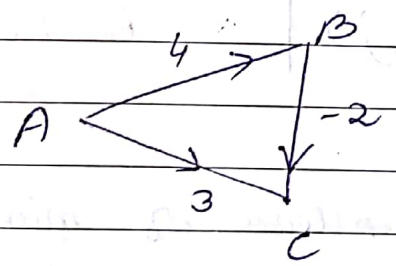
$p(5) = 5 + 6 = 11$

path $\longrightarrow$ print(v)

if $(v \neq \pi(v))$ print $\pi(v)$

↳ if weights are not positive, dijikstra's algorithm does not compute shortest path.

eg:-



$P(A) = 0$

$P(B) = 4$

$P(C) = 3$

C was already deleted, saying $A \to B = 4$ (correct)

$A \to C = 3$ (false

$A \to B \to C = 2$.

Proof:-

- shortest path depends on priority.
- once vertex ~~depends~~ deletes, priority does not change
- when in Q, priority can be decreased

$s, v_1, v_2, \ldots \quad v_{n-1}$ are vertices deleted in above order

$\longrightarrow$

priority is increasing order.

$p(s) \le p(v_1) \le p(v_2) \le \dots \le p(v_{n-1})$

↓

it is true, proof by induction.

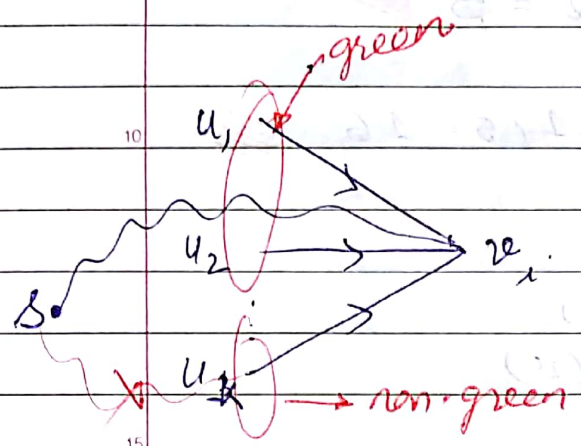
$\le p(v_i) \le \underbrace{\le p(v_j) \le p(v_{n-1})}$

↓

priority may decrease, but it will not decrease below $p(v_i)$

↓

if negative edge is there, it may be isolated so, to avoid such a condition, consider only +ve wts.

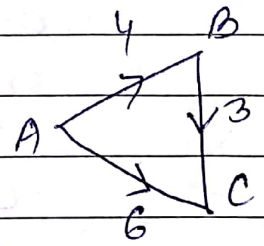


atleast one of $u_1, u_2, \dots u_k$ is green (deleted already)

$p(v_i) = p(v_j) + wt(i, j)$

Binary Heap → $O((V+E)\log V)$
Fib. Heap → $O(E + V\log V)$

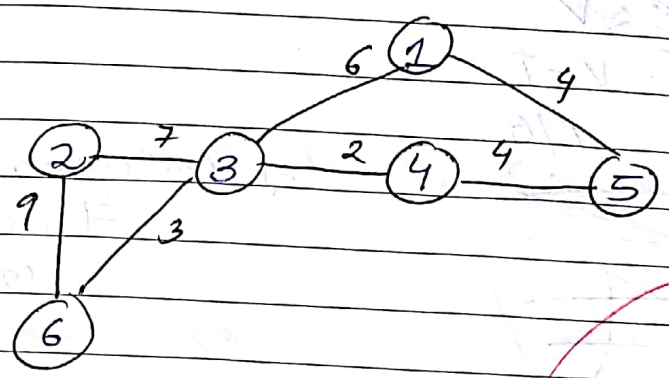
↳ <u>Longest path</u> :- instead of min PQ, use max PQ



$P(A) = 0$ ✓
$P(C) = 6$ ✓✓
$P(B) = 4$ ✓✓

should be 7. so, it cannot be found even if there is no cycle.

eg:- undirected graph



$P(s) = 0$
enqueue (S)
color(s) = Blue
$\pi(s) = s$
while (Q)
u = delete_min (Q)

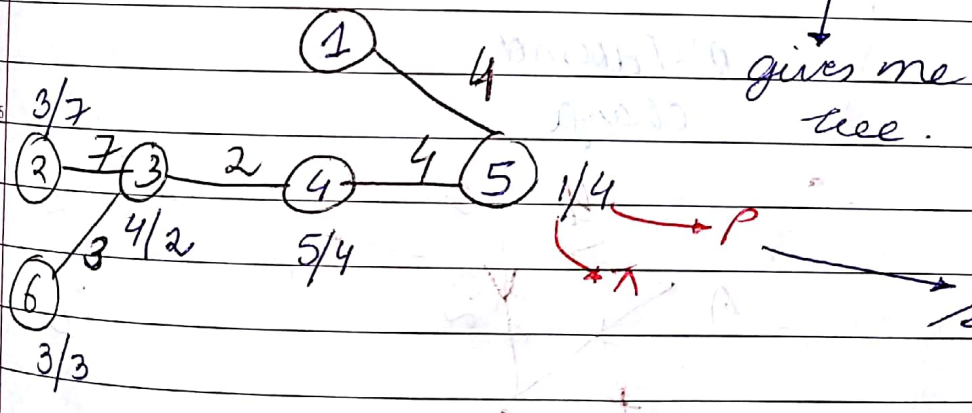
u
v
if (C(V) == Red)
$P(v) = wt(u,v)$
enqueue (v)
$\pi(v) = u$
$C(v) = B$
else
if (C(V) == B)
& $P(v) > wt(u,v)$
$P(v) = wt(u,v)$
$\pi(v) = u$

$(\pi(v), v) \in E$

(n-1) edges will be there except for first one if graph is connected.

(Prim's algorithm)

gives me a spanning tree.



minimum spanning tree

If, I use max PQ, then I get maximum spanning tree

cut in the graph

$V = S \cup T$ → not picked

$S \leq V$

$T = V - T$

if optimum < MST
∃ one edge (atleast) whose wt is < edge in MST.

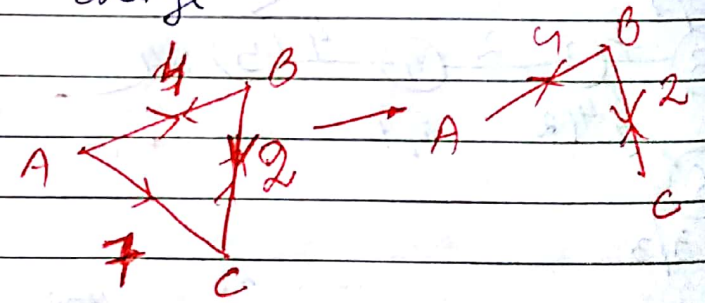
→ Shortest Path (SP) , Spanning Tree (ST)

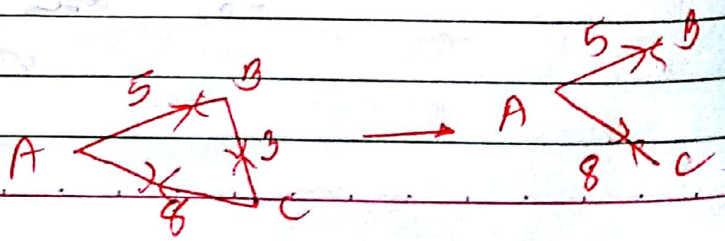
(i) $(\pi(v), v) \in E$

SP → Tree
ST → Tree } these trees may not be same.

(ii)

	G	G'	G''
	w	$w' = w + 1$	$w'' = 2w$
SP	may change		not change.
MST	$\sum_{v \in V} \pi(v), v) = n-1$ ↓ MST does not change		does not change.





LECTURE-29

- **DFS :**

(Recursive code)

source Node is not visited)

```
DFS (G)
    ∀v ∈ V ,   c(S) = R,   π(v) = NIL
    time = 1
    DFS_visit(S) ,   π(S) = S
    ∀v ∈ V ,  (if color(v) == R)
                DFS_visit(v)
```

in the queue.

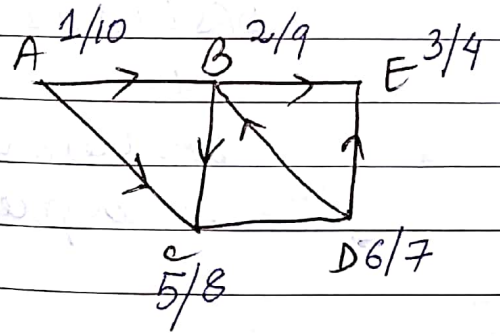
```
DFS_visit(u):
    color(u) = Blue
    d(u) = time++;
    ∀ (u,v) ∈ E,   if (color(v) == Red)
                π(v) = u
                DFS_visit(v)
    color(u) = Green
    f(u) = time++;
```

discovery time

visited.

finish time

eg. -



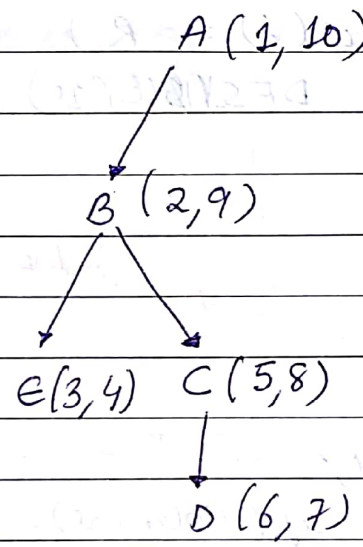
π	d/f
A → A	A → 1/10
B → A	B → 2/9
E → B	C → 5/8
C	D → 6/7
	E → 3/4

discovery time & finish time → $d \rightarrow (1, 20)$ in b/w
$f \rightarrow$

$d < f$

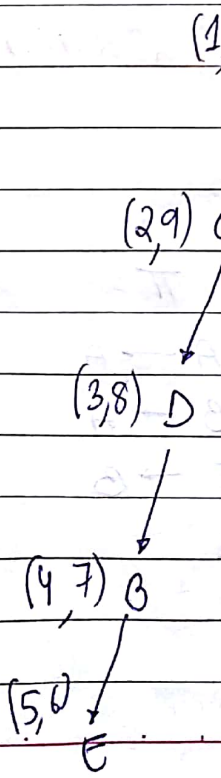
so, $\forall$ Vertices,
↳ $[d(v), f(v)]$

↳ $\pi(v)$ will give me a tree. → DFS Tree



it depends on adjacency list

OR



so, for the same graph if adjacency list representation is different, DFS trees are different.
i.e.
DFS trees are not unique

→ look at the edge which is outgoing from a vertex 'u' to vertex 'v'. if color ≠ green, change color to red, push to stack, look every edge outgoing from that.

↓

ensures I've visited all the edges reachable from this edge.

↳ when you delete a vertex first time, check discovery time. When you delete a vertex second time, check finish time.

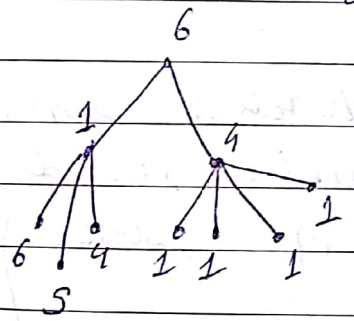
↳ figuring out whether I'm deleting for the first time, second time or third time? → use f(v)

if color == green & f(v) == 0, set f(v) to f(v)++.

If it is already set, don't do anything.

↳ <u>Applications of DFS</u>

(I) Give a tree. Every vertex has a weight associated with them.

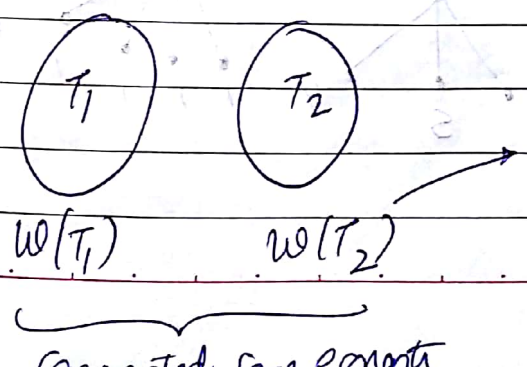


$T: w: v \longrightarrow R$ (no constraint whether +ve or negative)

n = # of vertices

$n-1$ edges in graph.

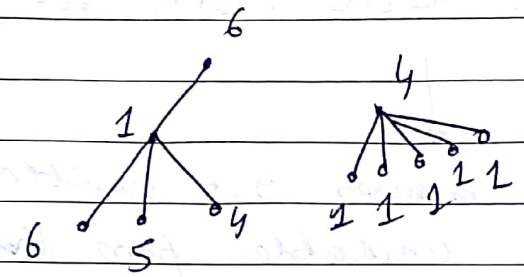
If I delete any one edge, I get two connected components.



→ sum of weights of elements in T_2.

I want $\boxed{\max(w(T_1), w(T_2))}$ ← I have this (I delete this bl my cost)
↳ minimum value

eg: →



(22) (9)
↳ cost of deleting an edge

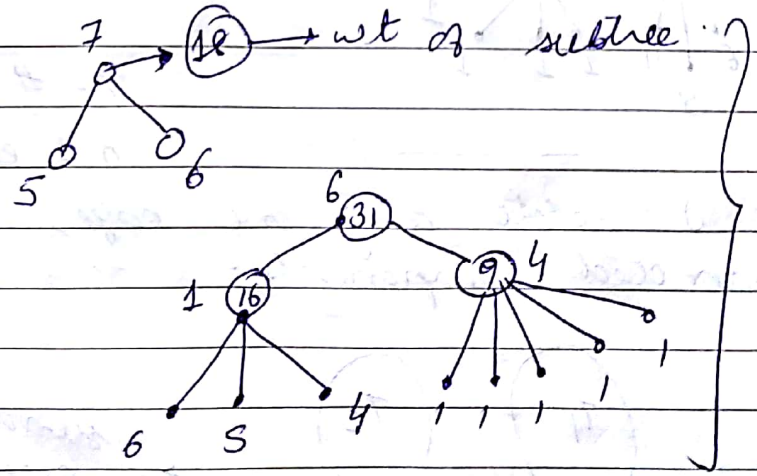
<u>brute force</u> :-

delete edge,
visit next vertices
estimate cost
} do this n-1 times
∴ T(n) for DFS/BFS
$= O(V+E)$
$= O(n)$

So, it is $O(n^2)$ algorithm.

I want max. of sum of elements in a connected component to be minimum.

↳ In a node, keep a track of sum of weights of the nodes in a subtree rooted there.



So,

this can be done when you finish your visit after DFS.

$O(V+E)$

~~DFS(G)~~ DFS_VISIT(G)

color(u) = B

d(u) = time++

$\forall (u,v) \in E$, if (c(u) == R)

$\pi(v) = u$

DFS_VISIT(v)

color(u) = Green

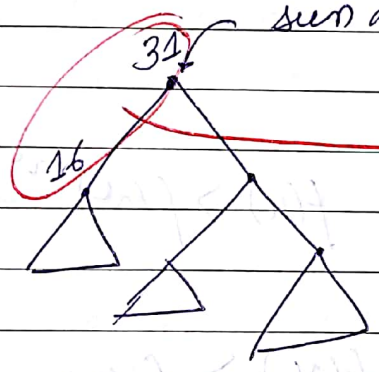
f(u) = time++ $\quad n(u) = w(u)$

$\forall (u,v)$ $\quad n(u) = n(u) + n(v)$

if (min(max (n(u), w - n(u))) min -

in DFS(G), set n(u) = 0 before the recursive calls in the loop. and also $w = \Sigma n(u)$

also, ~~max~~ max = w - ~~min~~ → sum of every vertex in a node



when you delete this edge, you can figure out, in one connected component, your sum = 16, other will be 31 - 16 = 15

Compute min(max(w(T), w(T(v))))

and so on, repeat for all nodes except root.

v: [d(v), f(v)]

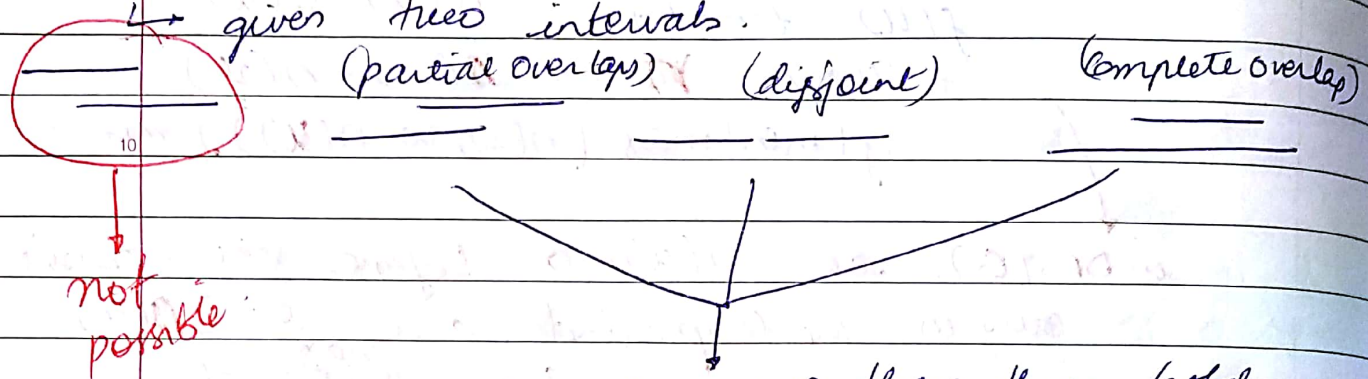
(G, d, f) → v

distinct (2n numbers)

$f(u) < f(v)$ always.

if $f(v_1) > f(v_2)$
$\Rightarrow$ v_1 is ancestor of v_2
and so there must be a path in reverse direction.

↳ gives two intervals.

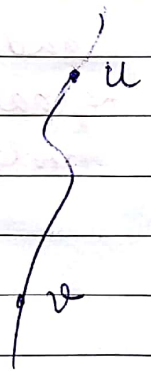


anyone of these three holds

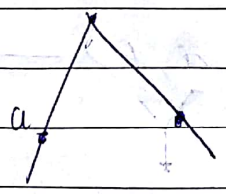
↳ <u>Theorem</u>.

$u \rightsquigarrow v$ $\quad f(u) > f(v)$ $\quad v \rightsquigarrow u$

$u \rightsquigarrow v$ $\quad f(v) > f(u)$ $\quad v \longrightarrow u$



along the same path $\begin{cases} d(u) < d(v) \\ \text{then} \\ f(u) > f(v) \end{cases}$



$\left.\begin{array}{c} d(u) < d(v) \\ \text{and} \\ f(u) < f(v) \end{array}\right\}$ on different paths

Directed Acyclic Graph (DAG)

arrange vertices in decreasing order of finish time.

$O(V+E)$ $\quad f(v_1) > f(v_2) > f(v_3) > \cdots > f(n)$

$\downarrow$

$O(V+E)$ $\quad$ I can get this by maintaining an array or linked list.

$\downarrow$

every time you finish something add it in the end of the list.

v_n	..	v_2	v_1

Topological sort :-

in this arrangement of vertices there is only a forward edge.

i.e. v_i is connected to v_j iff $j>i$

i.e. $\rightarrow$ if $i<j$, then $(v_j, v_i) \notin$ Edge.

(in decreasing order of f_i)

$v_1, \; v_2, \; v_3, \; \ldots v_i, \; \ldots \; v_j, \; \ldots \; v_n$

(suppose)

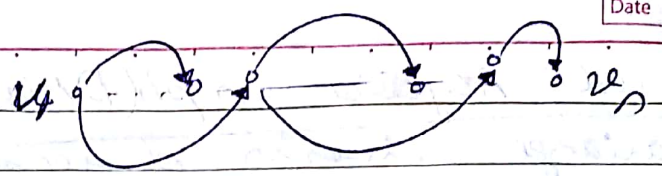
$f(v_i) > f(v_j)$

but if this edge is there, then there has to be a path b/w v_i and v_j in forward direction

$\Rightarrow$ cycle exists!

Cycle is not possible in above sequence (DAG)

(II) ↳ DAG :-



$d[\,] = \{\infty\}$

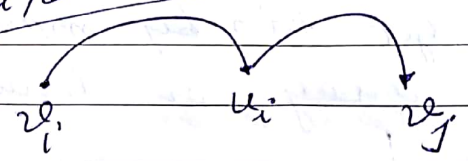
look at $(v_i, v_j) \in E$

if $d(v_j) > d(v_i) + w(v_i, v_j)$

$d(v_j) = d(v_i) + w(v_i, v_j)$

$\pi(v_j) = v_i$

(shortest path in DAG)



it works as there are no cycles & only forward edges.

↳ all incoming edges to a vertex is on the left of it.

$$T(n) = \begin{cases} O(V+E) & \longrightarrow \text{to find array element order (DFS)} \\ + & \\ O(V+E) & \longrightarrow \text{DAG} \end{cases}$$

↳ (longest path in DAG)

$d[\,] = \{-\infty\}$

↳ $\forall (v_i, v_j) \in E$

if $d(v_j) < d(v_i) + w(v_i, v_j)$

$d(v_j) = d(v_i) + w(v_i, v_j)$

$\pi(v_j) = v_i$

(III) Given a graph, find strongly connected components

↳ I want to make disjoint union of vertices.

disjoint union.

$$V = V_1 \dot\cup V_2 \dot\cup V_3 \dot\cup \cdots \dot\cup V_k$$

$G(V, E)$,

$G^T = (V, E^T)$ $\quad (u, v) \in E \iff (v, u) \in E^T$

↓

Transpose of a graph.

↳ in case of adjacency matrix, transpose of graph = transpose of matrix.

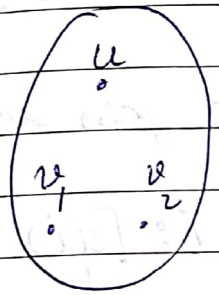
↳ it can be done in $O(V+E)$ using adjacency list.

↳ apply DFS and arrange vertices in decreasing order of finish time.

$v_1 \ldots v_n$ $\left\{ \begin{array}{c} f(v_i) > f(v_j) \\ i < j \end{array} \right\}$

↳ Take transpose of the graph G^T, (V, E^T)
apply DFS to transpose.

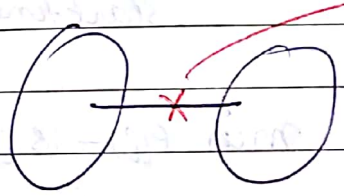
theorem done earlier



① $u \rightsquigarrow v_1$ in G^T

② $u \rightsquigarrow v_2$ in G^T.

$v_1 \rightsquigarrow u$ in G $\quad f(u) > f(v_1)$ $\quad u \rightarrow v_1$



if this edge was here, it would have been a single connected component

$u \rightsquigarrow v_1$ in G

$O(V+E)$

③ $v_2 \rightsquigarrow u$ in G

$f(u) > f(v_1)$

④ $u \longrightarrow v_2$ in G

LECTURE 30

Greedy algorithm :-

Greedy algorithm is an algorithmic technique to solve optimization problems.

- always makes the choice that looks best at the moment.
- makes a locally optimal choice in hope that it will lead to a globally optimal soln.
- it does not work for all problems.

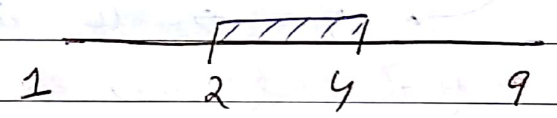
(I) Job Scheduling :-

Given a set of jobs with start time & end time, find a schedule on a m/c.

Object :- schedule maximum # of jobs w/o overlap.

eg: [2, 4]

$i : [s(i), e(i)]$



start time, end time.

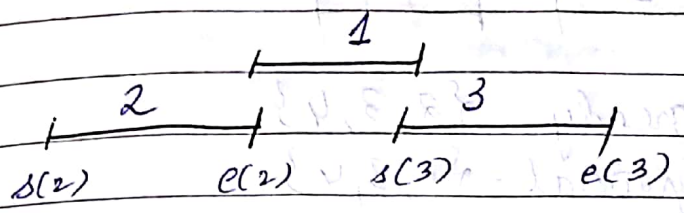
first job to pick → min $e(i) - \cancel{(s\ i)}$ → SJF

```
while (S)
    S = S - all jobs that overlaps with 'i'
```

$s(i) \le e(j)$
$s(j) \le e(i)$
} overlapping condition

pick the job which has min # of overlaps.

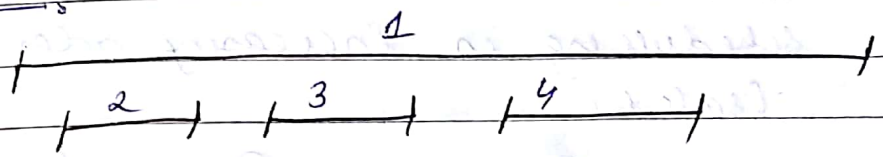
↳ Shortest Job First → does not give optimum



job 1 = shortest.
If you pick 1, you cannot pick 2 & 3

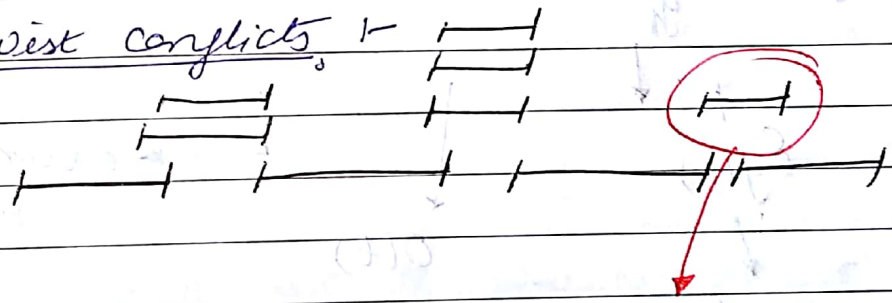
Greedy → {1}
optimum → {2} ∪ {3} = {2, 3}

↳ FCFS :-



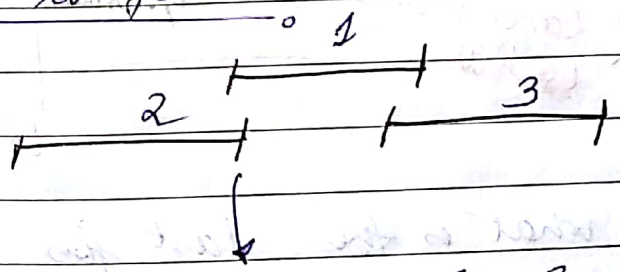
Greedy → {1}
optimum = {2, 3, 4}

↳ Fewest conflicts :-

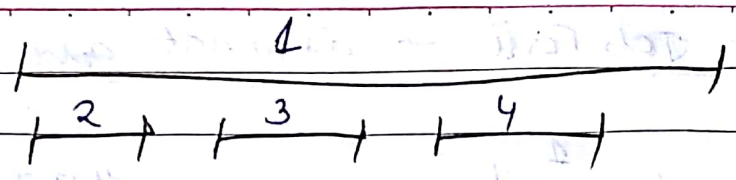


greedy = {x, y, z} = 3
optimal = {a, b, c, d} = 4

↳ First to finish



greedy = {2, 3}
optimal = {2, 3}



greedy = {2, 3, 4}
optimal = {2, 3, 4}

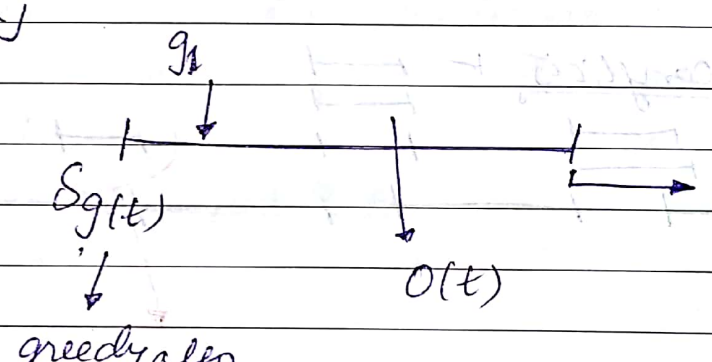
Proof:-

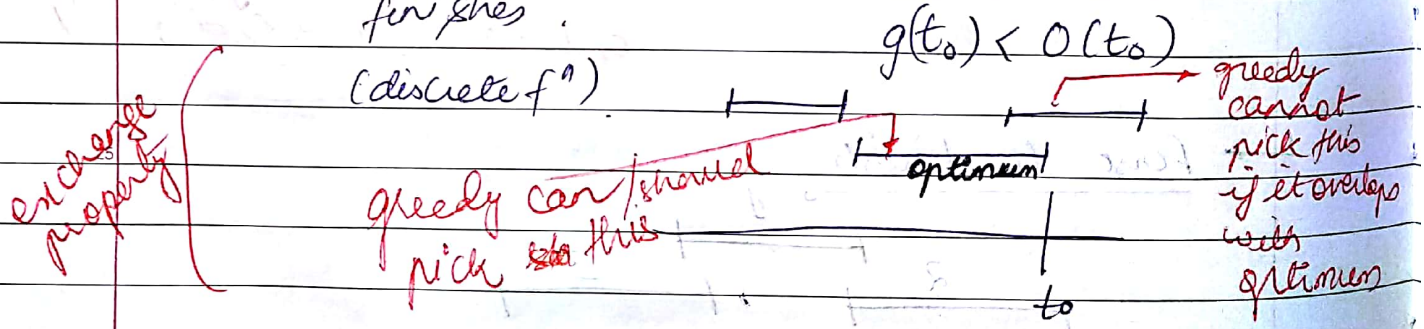
not optimum
⇒ there is an OPT sol^n with jobs > Greedy

scheduling in increasing order of finish time (sorted)

g_1 - - - - g_k (greedy jobs)

$e(o_i) < e(o_j)$ o_1 - — o_m (optimum)
$i < j$ $m > k$.





before t_0, what is the last job greedy did.
↓
should have finished before $s(i)$ of optimum

$O(n \log n + n)$

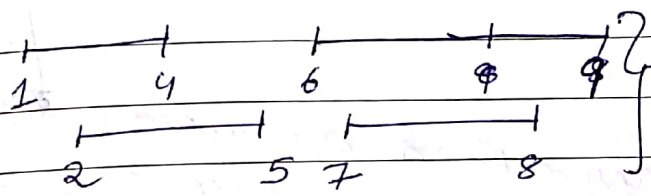
sort — traversal

(II) Schedule All jobs :-

↳ given jobs have a start time & end time

↳ schedule all jobs using min # of m/c's

max K conflicts ⟶ min 'K' m/c's required



1 — 4
6 — 9
2 — 5
7 — 8

1 — 4
2 — 5
6 — 9
7 — 8

2 m/c's reqd

A: 1, 3

B: 2, 4

↳ Depth :- of a set of jobs is the maximum # of jobs that are in conflict at any time.

↳ Greedy here is optimum.

(i) Sort on start time. (increasing order)

(ii) allocate new m/c only when required, needs only 'd' m/c's, where 'd' is depth of jobs.

↓

online algorithm

(don't know when it will come, we know only when it comes)

proof :-

→ assume greedy ends up using 'd+1' m/cs

→ at some time 't', it starts using (d+1)th m/c for i th job

1

2

3

.

.

.

.

d

d+1

$s(j) \le s(i) \le e(i)$

$s(i) \le e(j)$

if not true,

i

→ any of the d m/cs above should have jobs whose s_i < $s(i)$ of d+1th m/c

→ (d+1) jobs overlapping

Contradiction!

we need to maintain end time of jobs assigned to the m/c

$O(nk)$

→ k m/cs

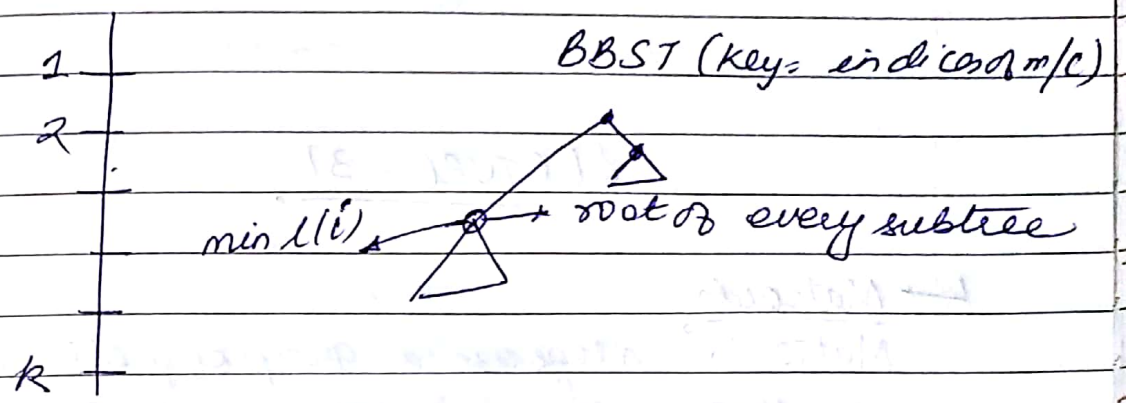
# of m/c maybe n,

so $O(n^2)$.

```
for i = 0 to K
        if l(j) < S(i)
            { l(j) = e(i)
              break;
            }
    l[k+1] = e(i)
```

↳ Build a minheap of $L(i)$ → find min $l(i)$ if $l(j) < s(i)$, then add.
$O(n \log n)$.

↳ depts



(II) Task scheduling Problem →
A set of unit jobs to be scheduled on a m/c with deadlines. If you miss the deadline, you need to pay penalty. Obj:- minimize penalty

eg:

4	2	4	3	1	4	6	7	deadline
70	20	30	50	80	20	10	5	penalty

↳ earliest deadline first → not a good strategy.

deadline	1	2	2
penalty.	1	10	10

↳ maximum penalty :- consider jobs in descending order of penalties & schedule as late as possible.

deadline	1	2	2
penalty	10	1	1

$O(n^2)$
↓
worst case.

Data Structure :- F F F

1st location to left of it

eg: 0 1 0 1 0 1 0 0 0 1 1 1 0 1

this one

LECTURE - 31

Matroids

Matroids categorize a group of problems for which greedy algorithm yields optimal solution.

Matroid:-

Let 'S' be a finite set $|S| = n = \{1, 2, \ldots n\}$ and a family of subsets 'F' (called independent sets) of 'S'

$$|S| = n$$
$$P \subseteq P(S)$$
$$|F| = m \quad \leq 2^n$$

eg:- $S = \{1, 2, 3\}$
$P = \{ \phi, \{1, 2\}, \{2, 3\} \}$

(I) Let S be a finite set and $F \subseteq P(S)$
We call (S, F) a matroid iff :-

M0) $\phi \in F$

M1) If $B \in F$ and $A \subset B$, then $A \in F$
[the family 'F' is called Hereditary]

M2] If $A, B \in F$ and $|A| < |B|$ then there exist x in $B \setminus A$ such that $A \cup \{x\} \in F$.
[this is called exchange property]

eg:- $A = \{2, 3\}$
$B = \{3\}$ $\quad |A| > |B|$

I need an $\{x\} \in A$ but $\{x\} \notin B$.

$B \cup \{2\} = \{A\} \in F$

↳ Weighted matroids :-

A matroid (S, F) is called weighted if it is equipped with a weight function $w: S \longrightarrow R^+$, i.e. all weights are positive real numbers.

If A is a subset of S, then
$w(A) := \sum_{a \in A} w(a)$

The problem is to find an independent set with maximum weight.

eg:- $S = \{1, 2, 3\}$
$\downarrow$
wts: $\{5, 7, 3\}$ $\quad w(A) = w(1) + w(3)$
$= 5 + 3 = 8$

using this algo, $O(m)$ time to find max
but $m \leq 2^n$

$f(A) = \begin{cases} \text{True} \\ \text{False} \end{cases}$ } I want such an algorithm independent of 'm'.

↳ Task scheduling problem :-

	4	2	4	3	1	4	6	7
Penalty →	70	20	30	50	80	20	10	5
A →	1	2	3	4	5	6	7	8

$S = \{1, 2, 3, \ldots n\}$ (set of all jobs).

$F = \{ A \subseteq S \mid$ all jobs in A can be scheduled before deadline $\}$

↳ order does not matter, schedule as late as possible avoiding penalty.

↳ <u>greedy algorithm for matroids</u>.

Greedy $(M = (S, F), w)$
$A = \phi$
Sort S into monotonically decreasing order by weight wt.
for each x in S taken in monotonically decreasing order, do:
if $A \cup \{x\}$ in F then $A = A \cup \{x\}$
return A.

$$T(n) = O\left(\underbrace{n \log n}_{\text{sorting}} + n.\underbrace{O f(n)}_{\text{can be } n,\ 1,\ \log n \text{ etc.}}\right)$$

↳ <u>Correctness</u>:

<u>Theorem</u> :- Let $M = (S, F)$ be a weighted matroid ~~with~~ function 'w'. Then Greedy (M, w) returns a set F of maximal weights

$$w(x_1) \geq w(x_2) \geq \cdots w(x_n)$$

$w(G) =$

greedy picks these.

$O :- \quad O_1 \quad O_2 \quad \cdots \quad O_{k_1} \rightarrow$ optimum

↳ $G :- \quad g_1 \quad g_2 \quad \cdots \quad g_{k_2} \rightarrow$ greedy

<u>assume</u>: $\exists$ set, $w(O) > w(G)$

$$\equiv \sum w(O_i) \qquad \equiv \sum w(g_i)$$

$w(O_i) \geq w(O_{i+1})$
$w(g_i) \geq w(g_{i+1})$

prove $\longrightarrow$ $K_1 = K_2$

$|O| < |G|$ $\qquad$ $|O| > |G|$

I cannot add in any of them, as weight of both will increase which contradicts.

so, $|O| = |G|$ i.e. $K_1 = K_2$.

now we compare, (both are sorted in decreasing order).

if $|O| > |G|$ then for one $|O_i| > |g_i|$

O': $O_1 \quad O_2 \quad \cdots \quad O_{i-1} \; O_i$
G': $g_1 \quad g_2 \quad \cdots \quad g_{i-1}$ } both independent

$|O'| > |G'|$

but $w(O_j) \geq w(O_i) > w(g_i)$ $\quad \exists O_j,\ j < i,\ \in O' - G'$ st $O_j \cup G' \subset F$
$\Rightarrow w(O_j) > w(g_i)$

(contradiction)

both sets should have same # of element.

↳ analysis :- the for loop iterates n-times. In the body of the loop one needs to check whether $A \cup \{x\}$ is in F
Thus greedy takes $O(n \log n + \phi . f(n))$ time.

↳ It was about maximum, what if I want minimum?

$$\min_{A \subseteq F} w(A)$$

sort in increasing order of wt & apply same algorithm.

$$w : S \rightarrow R^+$$
$$M = \max(w(x)) \quad x \in S$$
$$\left.\begin{array}{l} w'(x) = M - w(x) \\ w'(x) \geq 0 \end{array}\right\} \text{ for -ve weights.}$$

↳ weight matters only before applying algo, (to sort).

↳ After that what matters is, whether it is independent set or not; i.e. $A \cup \{x\}$,

$$N = \min(w')$$
$$w'(x) = w(x) + N + 1.$$

eg: $-3, 5, -6, 2, 8 \rightarrow$ -ve wts

$+6+1 \downarrow$

$4, 12, 1, 9, 15 \rightarrow$ all +ve wts

or

skip -ve wts.

any independent set with -ve wts

$\downarrow$

max wt will never have -ve elements.

↳ if wts are only +ve, apply the algorithm.

$|A| > |B|$ pick $x \in A - B$

set of jobs.

$A' = A - A \cap B$
$B' = B - A \cap B$

to show:- $\exists x \in A'$ st $B' \cup \{x\}$ is independent.

$x \in A' \subseteq A$.

A' ✓ X X ✓ X ✓ ✓ X ✓ ✓
B' ✓ X X X ✓ X ✓ ✓ X ✓

no intersection jobs

∴ # of elements in $A' > |B'|$
there should be a day when A' is doing something but B' is not doing anything.

A. ✓ X X ✓ ✓ X ✓
B: X

B might be doing the same jobs on some other day.

↳ <u>Spanning Trees</u>:- ST of a graph $G = (V, E)$ is a graph that satisfies:-
↳ no cycles (acyclic)
↳ no isolated nodes (connected)
↳ all nodes are present in tree.

- contain only those edges what are needed to connect all nodes.

$S = E$.

$F = \{$ set of all edges that do not form a cycle $\}$

↳ sort edges in increasing order of wts.
↳ select edges which are not forming cycle } Kruskal Algorithm

$O(E \log E + E.V)$ → DFS

↓ sort

but prim $= O(E \log V)$

we may not ignore -ve weights, if I ignore, I may not get spanning tree.

adding some constant to all edges, spanning tree does not change.

(continued)

LECTURE - 32

↳ **Kruskal** :- $G = (V, E)$

$S = E$

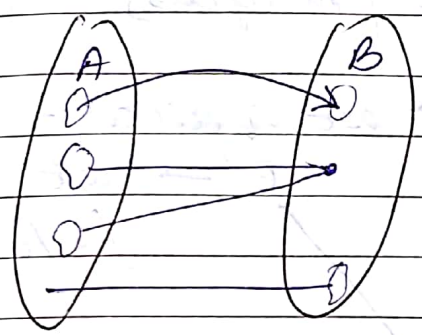
$F = \{A \subseteq E \mid$ the edges do not form cycle$\}$.

$M(S, F)$

0.) $\phi \in F$

1) $A \subseteq B$, $B \in F \Rightarrow A \in F$.

2)



$|A| > |B|$

$n(A) < n(B) \rightarrow$ # of connected components.

to prove:-

$\exists (u, v) \in A - B$

$B \cup \{u, v\}$.

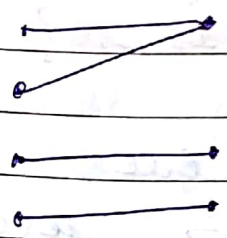
edge that goes across the connected components.

$\forall (u, v) \in A \setminus B$

$B \cup \{u, v\} \notin F$.

$|X| \geq |Y|$

X Y



connected components

$n(A) > n(B)$

contradiction

onto

$\exists e \in A \rightarrow B \cup \{e\} \in F$.

↳ method :- $M(S, F)$

$w : S \rightarrow R^+$

sort elements in 'S' in ~~increasing~~ decreasing order.

$A = \phi$

if $A \cup \{x\} \in F$, $A = A \cup \{x\}$

A is the independent set with ~~min.~~ max. wt.

$G = (V, E)$

$w : E \rightarrow R^+$

$S = E$

$F = \{A \subset E \mid \text{no cycles}\}$

→ wt of all edges is +ve, so, $E' \rightarrow$ wt = maximum (w/o cycle).

ST? if yes, maximum spanning tree.

What is the problem with minimum?

I may not take any edge, wt = 0.

add an edge → wt increases so I do not select any edge.

How will I get rid of this +ve?

for minimum spanning tree,

↳ negate the edges, sort in increasing order & repeat.

↳ so I get minimum spanning tree

But in practice, I'll not do this!

I'll sort and consider the sequence in reverse order, instead of negation.

↳ if you have everything (+ve & -ves), sort in decreasing order → max. ST.
ascending → min ST.

sort → $O(E \log E)$
Consider each edge → $O(E)$
whether to add edge or not → E times, it happens

maintain a collection.
new edge comes, apply DFS/BFS.

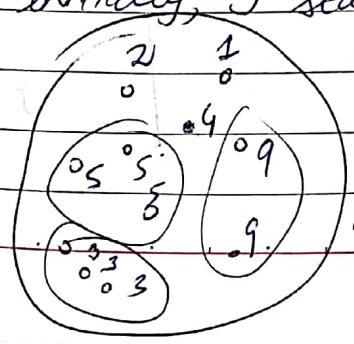
$O(EV)$ XX poor implementation

$f(A)$ → True (independent)
→ False (dependent).

$f(A \cup \{e\})$ is what I'm interested in.

Data structure required :- E (queries) edges, and exactly V-1 merges I'll be doing. to obtain spanning

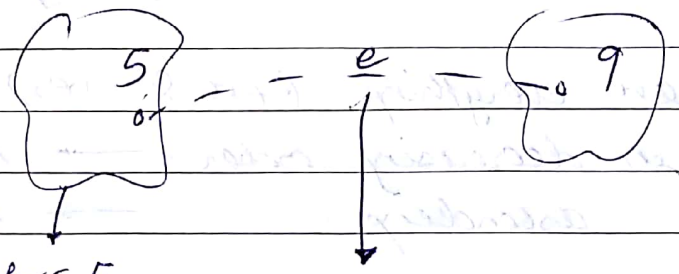
initially, I start with all isolated vertices.



↳ all edges in a connected component have same color.
↳ edge going across two colors is edge connecting them. merge them to form one connected component.

if color is same, neglect the edge (within same connected component, will form cycle).

$A = A \cup \{e\}$ if color is different



Change color of 5 to 9
or
change color of 9 to 5 } Better?

smaller component merges with bigger

every vertex is a structure
(I) v.color → enough to tell whether it goes from one connected component to other or not.
(II) array list 'N'
(III) Linked list 'L'

if (v_1.color == v_2.color) neglect edge.
if (v_1.color ≠ v_2.color)
& if ($N(C(u)) \leq N(C(v))$)
$N(C(v)) += N(C(u))$
$N(C(u)) = 0$ }

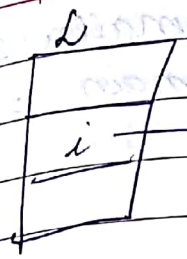
N

1 → 1
1 → 1
1
1

N(i) tells colored vertices number
$N(j) = N(i) + N(j)$
$N(i) = 0$

else, do opposite (merge 'u' with 'v')
~~N(C(v)) = 0.~~
}

(X)



→ list of all vertices with color 'i'

updation → change color of each vertex

↓

↳ goto each vertex,
↳ change color,
↳ add in individually to new place
↳ delete it & continue.

at the end → I'll have exactly one connected component

↓

I'm doing exactly one merge operation.

↓

(x)

while (L(N(v)))

[do above thing

(V-1) merges

if each merge = V/2 (worst case)

= $O(V^2)$.

but this happens only once. (last one)

total merges = log V.

but it happens only once.

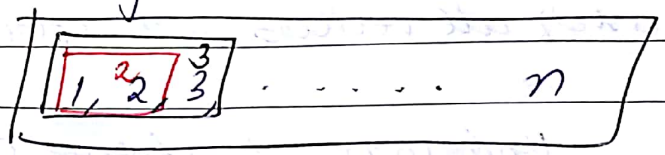
so,

$O(V \log V) + O(E) + O(E \log V)$

$= O((E+V) \log V)$

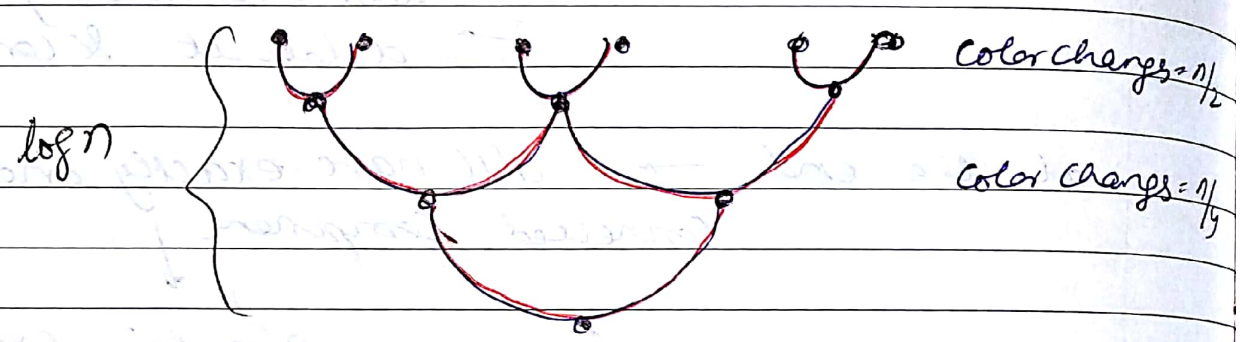
↳ same as prim's.

If we merge bigger set with smaller set, Complexity will continue to remain $O(V^2)$



$T(n) = (n-1) + T(n-1)$
$= O(n^2)$.

but merge smaller with bigger
$T(n) = O(n \log n)$



$\frac{n}{2} \log n$ → but not necessary

How many times can a vertex undergo color change?

Theorem :- $\forall v$, the no of times its color changes is at most $\log V$.

so total color changes for V vertices is $O(V \log V)$

⇒ sorting $= O(E \log V)$

so $T(n) = O((E+V) \log V)$.

so now, after 'k' - color changes, the # of elements in the connected component to which this vertex 'u' belongs to should be atleast 2^k.

$k = 0 \longrightarrow 2^0 = 1$ (isolated vertices)

$k = 1 \quad \geq 1+1 = 2^1$

$k \longrightarrow 2^{k-1}$ (after $k-1$ color changes by induction)

after k^{th} color change

it should be merged with set greater than this, otherwise color of this won't change

so $2^{k-1} + 2^{k-1}$ (atleast)

$= 2^k$

w/o this, it is not feasible.

$2^k \leq n$

$\Rightarrow$ max color changes

$= \underline{k \leq \log(n)}$

↳ Problems on minimum spanning tree :-

(I) $G(V, E)$ $\quad G^t$ } it is a dynamic graph

$MST = T$ $\quad |E|$ keeps increasing/decreasing

add edge $(u, v) \in E$

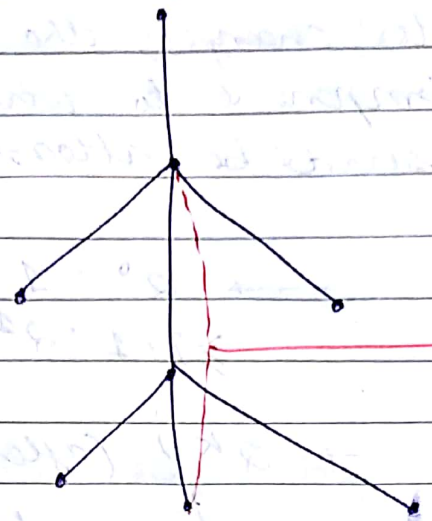
$w(u, v)$ = wt of this weight.

Update MST :-

apply it again $\rightarrow O((V+E)\log V)$

anything faster $\rightarrow$ take this edge, add to MST

$M(u,v)$ = max wt in the cycle created by adding $(u,v) \in E$ to the MST.



If I add this edge, there will be a cycle, so we need to delete one edge in the cycle, with maximum weight

↓

this will decrease weight of the MST.

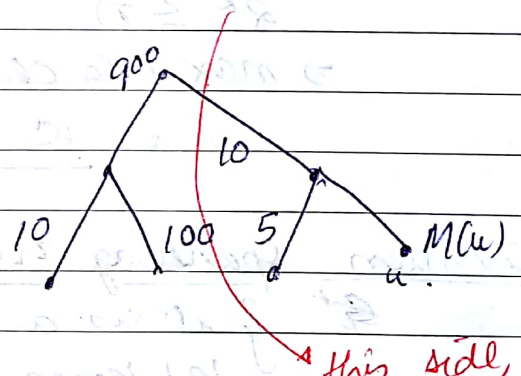
find max. wt edge b/w 'u' & 'v' in a cycle ?

↓

how fast can it be done

↓

DFS / BFS.



this side, 900 not be considered.

I'll maintain as $\underline{M(u)e}$.

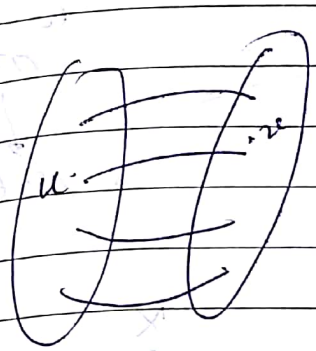
relax edge (u_1, u_2)

$O(V)$

$M(u_2) = \max(M(u_1), wt\ e(u_1 u_2))$

(II) $G(V,E)$, T, delete edge (u,v)
update ST.

if edge $\notin T$ → do nothing $O(1)$.
if edge $\in T$ → update tree,
it'll create a cut.



one component → color 1
2nd component → color 2

$O(E)$ ← add lowest wt edge that goes across opposite colors

↳ If sorted order is already given

updating → costlier..
add → $O(V)$
delete → $O(E)$.

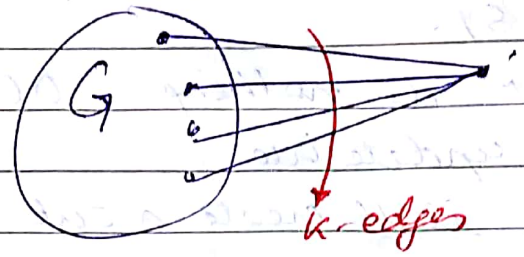
do something
$O(E)$
$\in$ tree ✗

(III) weight is changing now. → increases → $\notin$ tree ✗
→ decreases.
$\notin$ tree
$\in$ tree
do nothing

add to tree,
find max wt edge in cycle, delete it
$O(V)$

(IV) Can I do something to shortest path?
↳ much harder than updating ST.

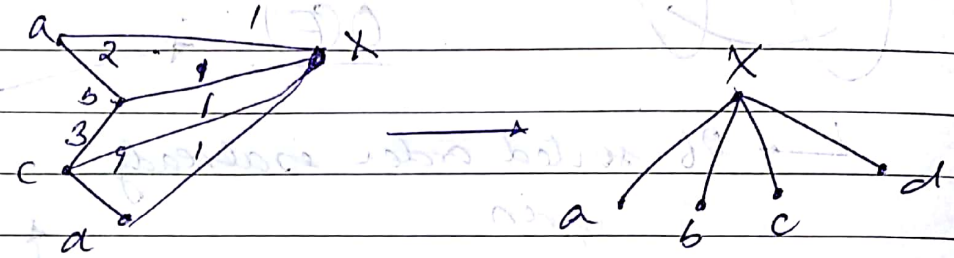
(V) $G(V, E)$, I want to add K - edges.



min (K-edges)
add it
will it be a MST?
Not certainly

if weight of all the edges is 1?

eg:



So, add one edge using add algorithm. remaining $K-1$ edges can be added using algorithm above.

if $K = O(n)$ $\xrightarrow{\text{it becomes}}$ $\underline{O(VK)}$ $\underline{O(V^2)}$.

$G' = T \cup \{K\}$

apply Kruskal / prim

almost $(n-1)$ edges

So, $|E| \leq 2n-2$
$\Rightarrow \underline{O(n \log n)}$.

if K is $< \log n$, apply $O(VK)$

if $K \geq \log n$, apply this

LECTURE-33

Network flow :-

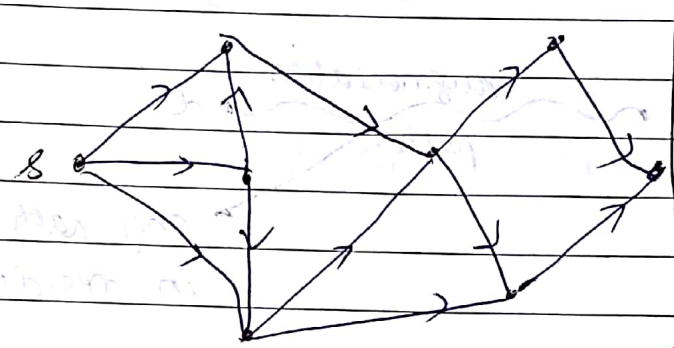
$G(V, E)$

Capacity 'c' : $E \rightarrow R^+ / Z^+$

flow 'f' : $E \rightarrow Z^+$

s = source

d/t → sink / destination



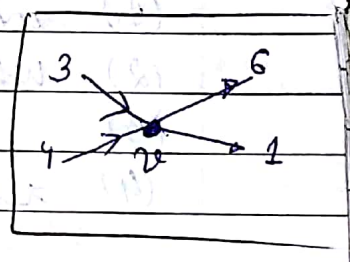
$$\max \sum_{(s,v) \in E} f(s,v) = \sum_{(v,d) \in E} f(v,d)$$

(1) $\forall (u,v) \in E, 0 \le f(u,v) \le c(u,v)$ → capacity constraint

(2) $\forall v \in V$, → flow control

↳ $\sum_{(u,v) \in E} f(u,v) = \sum_{(v,u) \in E} f(v,u)$

inflow outflow.



(3) $f(u,v) = -f(v,u)$

↳ Ford - Fulkerson's Method :-

It contains 3- ideas

↳ residual n/w

↳ augmentation paths

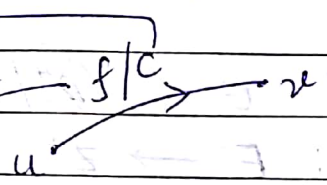
↳ cuts of the n/w flow.

↳ <u>residual n/w</u> $G(V, E) \longrightarrow G_r = (V, E_r)$

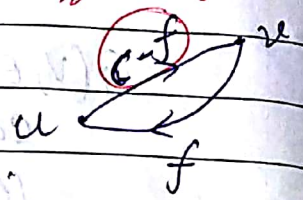
given $\begin{cases} f = 0 \\ c \end{cases}$

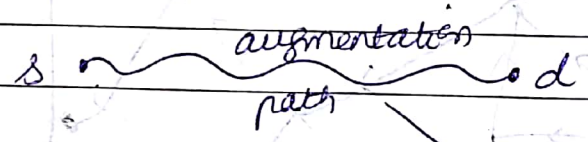
capacity of flow.

present flow



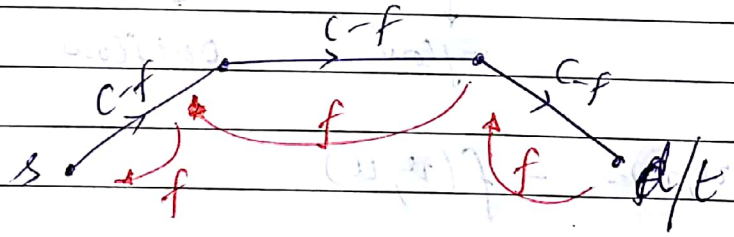
still send a flow of c-f





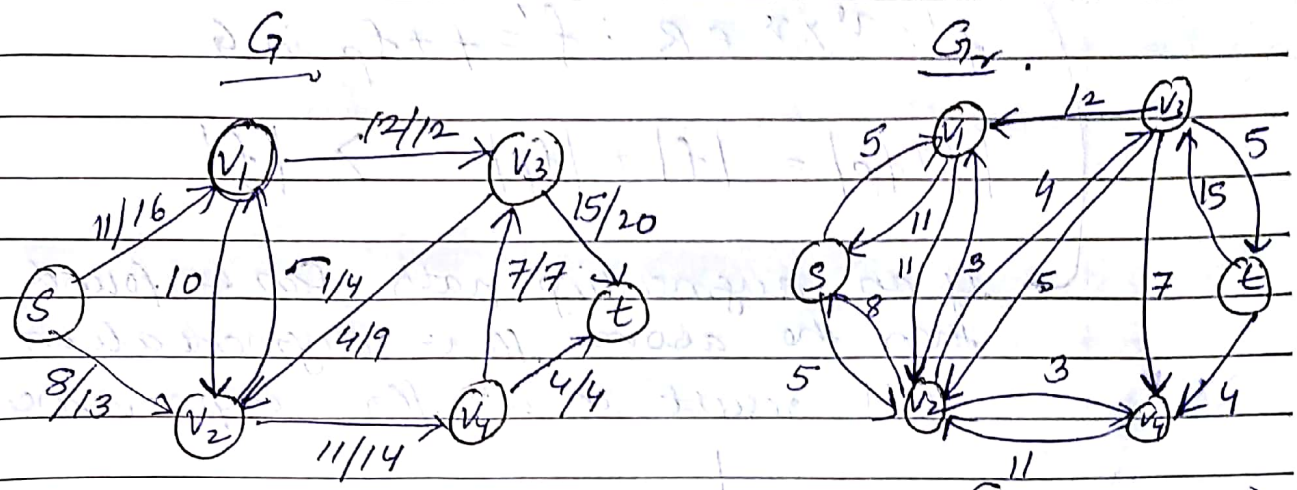
any path from s → d in residual graph.

(1.) initialize $f = 0$
(2) while $\exists$ p (an augmentation path)
(3) do augmentation flow 'f' along 'p'
(4) return f.



↳ The residual graph (G_r) of a given flow n/w G with a valid flow 'f' consists of the same vertices $v \in V$ as in G which are linked with residual edges $(u, v) \in E_r$ that can admit more strictly positive net flow. The residual capacity represents the wt of each edge E_r and is the amount of additional net flow $f(u,v)$ before.

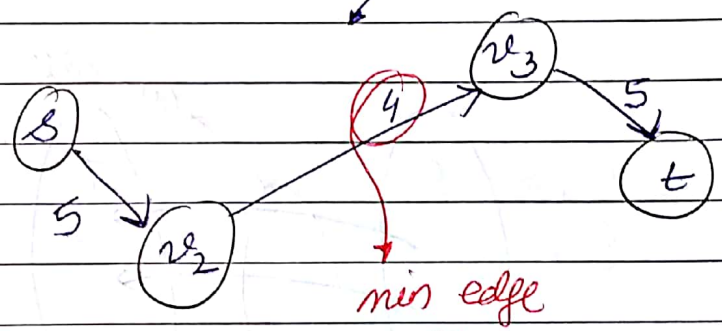
exceeding the capacity $c(u,v)$.



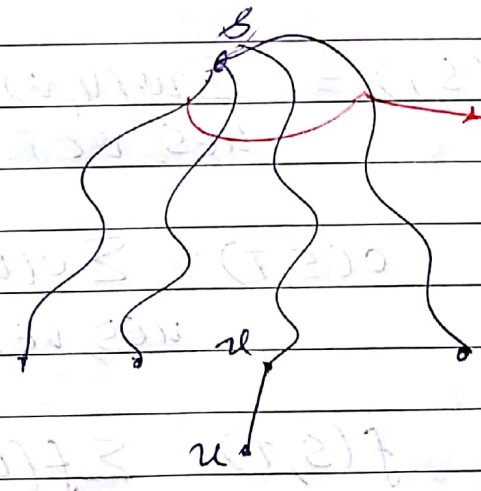
(augment ing path)

$$C_r(u,v) = C(u,v) - f(u,v)$$

Keep a track of $\pi(v)$
↓
to trace the path.



min edge wt = max flow possible



Various paths, where $M(v)$ shows/tells me wt of min wt edge along a path.

$$M(u) = \min(M(v), w(v,u))$$

Lemma :-

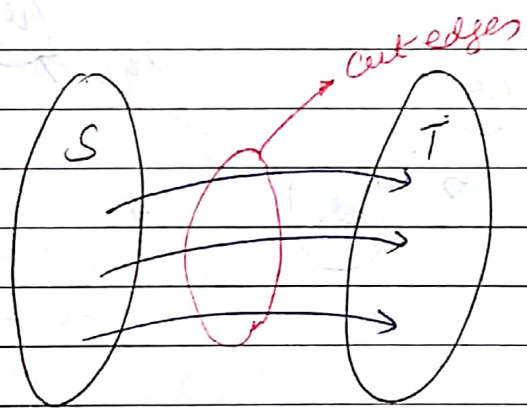
$f' : V \times V \in R : f' = f + f_p$ is G

$|f + f_p| = |f| + |f_p| > |f|$

→ if an augmenting path can be found, then the above flow augmentation will result in a flow improvement.

↓

If we cannot find any more augmentation path, is our flow maximum?



$V : S \dot{\cup} T$ → disjoint union

st cut,
$s \in S$
$t \in T$

$S \subseteq V : s \in S$
$T = V - S$

$$w(S,T) = \sum_{u \in S,\, v \in T} w(u,v)$$

$$c(S,T) = \sum_{u \in S,\, v \in T} c(u,v)$$

$$f(S,T) = \sum_{u \in S,\, v \in T} f(u,v)$$

Theorem :- $f(S,T) \le c(S,T)$

$$\sum f(u,v) \le \sum c(u,v) \quad , (u \in V, v \in V)$$

precisely $u \in S$, $v \in T$

↳ **Max flow, min cut theorem :-**

If 'f' is a flow in a n/w $G(V,E)$ with a source 's', and sink 't' or 'd' then following conditions are equivalent

↳ f is a maximum flow in G

↳ The residual n/w G_f contains no augmenting paths

↳ $|f| = c(S,T)$ for some cut $(S,T) \in G$

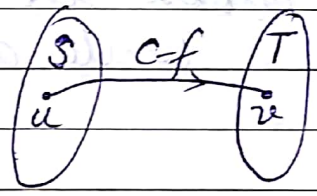
Proof :-

$2 \Rightarrow 3$. $\exists f(S,T) = c(S,T)$

$S = \{ v \mid s \rightsquigarrow v$ is residual n/w $\}$

$s \in S$

$t \notin S$



if $f(u,v) < c(u,v)$ there will be a residual n/w edge whose wt c-f will be +ve.

$$\sum_{(u,v) \in E} f(u,v) = \sum_{(u,v) \in E} c(u,v)$$

$$f(S,T) = c(S,T)$$

$3 \Rightarrow 1$ $\exists S,T$ st $f(S,T) = c(S,T)$

if f is not max, $\exists f'$, $f' > f$

$$f'(S,T) > f(S,T) = c(S,T)$$

but $f'(S,T) > c(S,T)$
which is not possible

<u>Complexity</u> :- $O(V+E)$ for 1 iteration
# of iteration = $O(|E| \cdot |f_{max}|)$

↳ any augmentation path is not ok for me, I need to put some restriction on augmentation paths.

so f' should be integer. If it is fractional, runtime, it may vary by a factor of 10^{-100}, so it'll be come untractable.

↳ <u>Edmond's Karp Algorithm</u> :-
choose the shortest length path while augmenting.

use BFS, $m(t)$ tells max. value.

one iteration = $O(V+E)$

But # of iterations is $O(VE)$

so, $T(n) = O(VE^2)$

<u>critical edge</u> :- (u,v) if

$$C_f(p) = C_f(u,v)$$

↳ atleast one edge on 'p' must be critical.

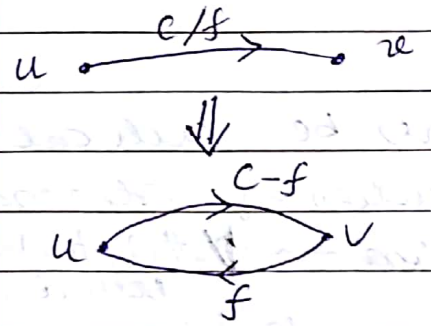
↳ # of times (u,v) be critical during execution of Edmonds-Karp Algorithm → ! # of times in both directions so $2V$.

↓

if the (u,v) becomes critical in the same direction, the length of the augmenting path is strictly smaller than what it was before.

LECTURE - 34

R.N. (residual n/w)



$G = (V, E)$

given $c : E \to Z^+$

find $f : E \to Z^+$

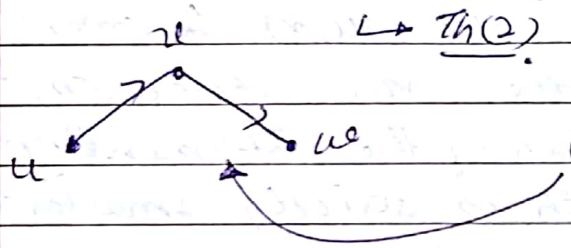
such that

$$\max \sum_{v \in V} f(s,v) = \sum_{v \in V} f(v,t)$$

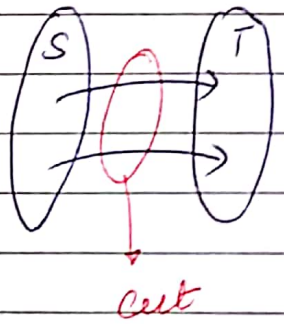
↳ Th(1) $f(u,v) \le c(u,v) \quad \forall (u,v) \in E$

↳ Th(2) $\forall v \in V - \{s, t\}$

$$\sum_{(u,v) \in E} f(u,v) = \sum_{(v,w) \in E} f(v,w)$$



↳ Residual flow R_f = min flow from $s \to t$



$V = S \dot{\cup} T \Rightarrow S \cap T = \phi$

$s \in S$

$t \in T$, where $T = V - S$

↳ Ford - Fulkerson algorithm :-

$f(S,T) \le C(S,T)$

flow — capacity

(1) fmax
(2) no aug. path
(3) $\exists (S,T)$ st $C(S,T) = f(s,t)$

$T(n) = O(V+E) \to$ 1 iteration

# of iteration = max flow

$S = \{ v \mid s \rightsquigarrow v \}$

all reachable from source.

↳ **Edmond-Karp's Algorithm** :-

find shortest length augmenting path.

$T(n) = O(VE^2)$, i.e., unlike Ford-Fulkerson, $T(n)$ does not depend on flow.

eg :- $G = (V, E)$ (changing)
$G' = (V, E)$

(I) → $c = (u, v)$ increased capacity

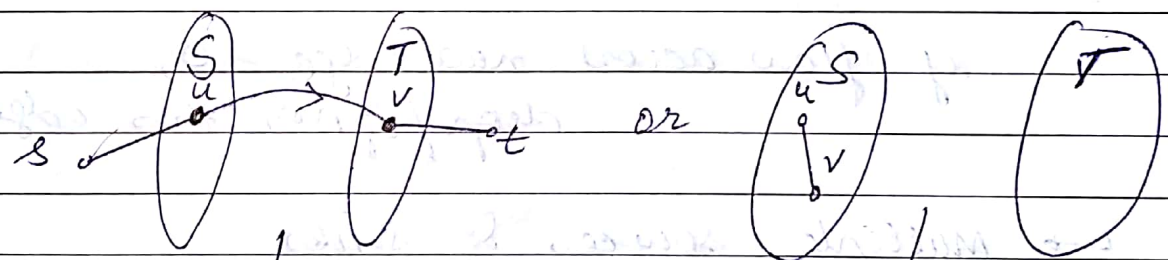
↓

$O(V+E)$
- ↳ compute RN.
- ↳ find augmentation path.
- ↳ If it exists, it should contain this edge.

Is 1-iteration enough? No!

↓

We may need to find more paths



This affects the flow

This does not effect the flow.

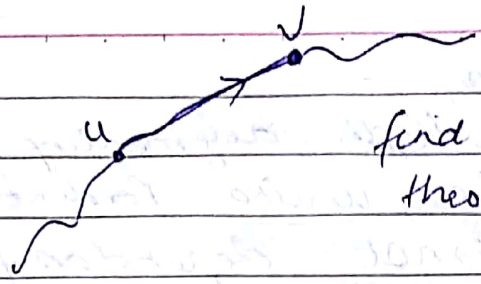
(II) $w(u,v) \downarrow$ (decreased capacity)

$f(u,v) \le c'(u,v)$ ideally

↓

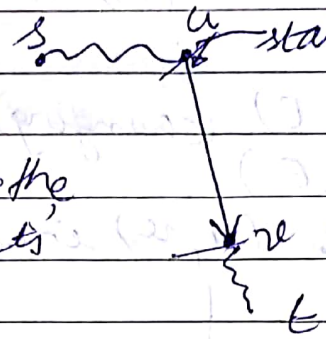
but now $f(u,v) > c'(u,v)$ → updated capacity

↳ so, f is reduced to $f(u,v) - c'(u,v)$



find augmenting path, which goes through this edge.

BFS/DFS



start BFS/DFS

find minflow along the path, decrease its value.

If $f \neq 0$, repeat.

(III) add edge → $c(u,v)$ → this edge was not there in previously computed network

↓

apply algorithm for increasing the capacity, retaining the old flow.

Consider, earlier capacity was '0' that is increased to $c(u,v)$

if flow across new edge = 0, drop/ignore this edge.

↳ Multiple sources & sinks

$f(u,v) \le c(u,v)$

$\forall v - \{s_1 \text{ to } s_{k_1}, t_1 \text{ to } t_{k_2}\}$

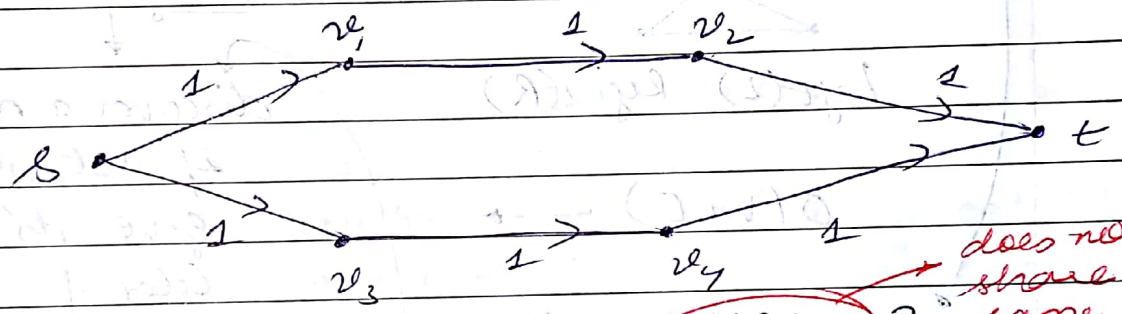
in flow = out flow.

$$\sum_{(s,v) \in \text{Edge}} f(s,v) \quad , \quad s \in \{s_1, s_2, \dots s_{k_1}\}$$

↳ add super source & super sink with infinite capacity.

↳ Applications

(1) edge disjoint S-T path. (unweighted graph)



how many paths are disjoint? → does not share same edge. It is different from vertex disjointness

Net flow → capacity?

↓

set capacity = 1 across all edges.

find max flow from s to t

↳ max # of disjoint paths.

(2) Delete min # of edges in the graph st there is no path from source to destination?

↓

delete all disjoint paths.

↳ Menger's Theorem :- (1927)

max disjoint paths = min # of edges to be deleted.

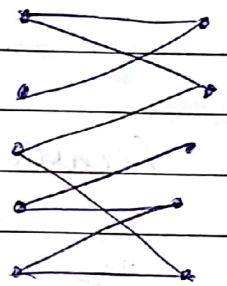
↓

follows from max cut, min flow theorem.

(3) Bipartite graph :- ↳ $V = L \cup R$.

↳ two colourable

↳ start at source 's', put at left, color = 0



Left(L) Right(R)

$O(V+E)$ →

also called stable marriage problem.

BFS

discover a new vertex, if not visited, give its' complement color.

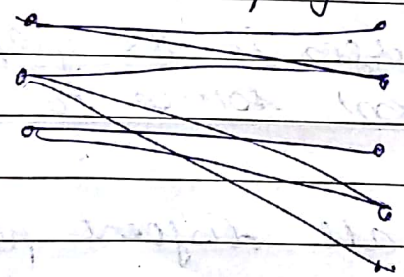
else,

check edge

Complement color → continue

same color → not Bipartite → stop.

eg:- students (S) projects (P)



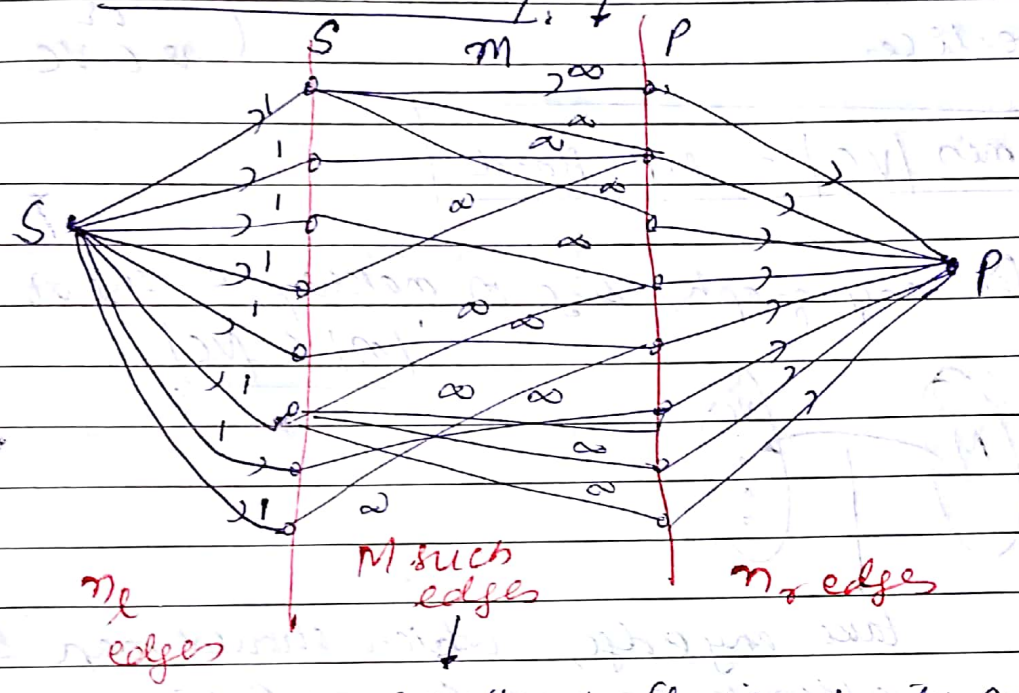
every project = 1 student

every student = 1 project

$G=(V,E)$, $M \subseteq E$ → do not have common end points.

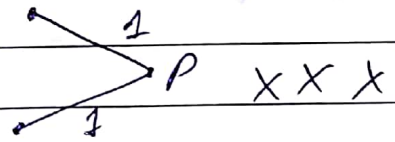
(M) max possible matching on a bipartite graph

maximum matching ↓



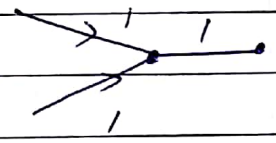
even though, flow is ∞, it cannot be more than 1, ($\xrightarrow{1} \cdot \xrightarrow{\infty} \cdot \xrightarrow{1}$)

max flow = max matching possible (M)



XXX

↳ look at optimum maximum matching



= $m \geq f$ ⟺ $f \geq m$

(end) start

$f = m$ = max # of disjoint paths.

$G=(V,E)$ max $|M|$ → Polynomial.

$M \subseteq E$ max matching $|M|$

$= \min(n_l, n_r)$

$|M| \le \min(n_l, n_r)$

↳ vertex cover :- $VC \subseteq V$ in $G(V, E)$

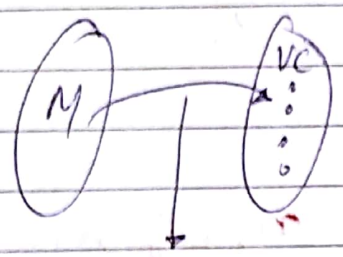
with min # of vertices.

$(u, v) \in E = \begin{cases} u \in VC \\ \text{or} \\ v \in VC \end{cases}$

$\boxed{\min |VC| = \text{NP Hard}}$

for any graph, size of matching $\le$ size of VC

$|M| \le |VC|$

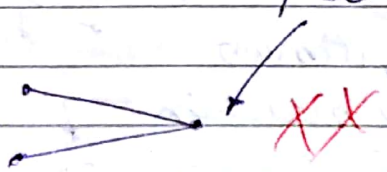


take any edge, which should map to atleast one vertex on RHS.

↳ This mapping is one to one (may not be onto)

$f(x) = f(y)$
$\Rightarrow x = y$

} two edges should map to different vertices.
i.e. no common end point.



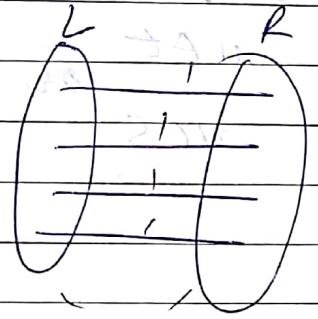
↳ In case of Bipartite graph
Konig - Egevari theorem
on a bipartite graph,
max matching $|M|$ = min vertex cover $|VC|$

$|M| = 1$
$|VC| = 2$ (min)

$1 < 2 \rightarrow$ so, theorem is not true for any general graph,

But

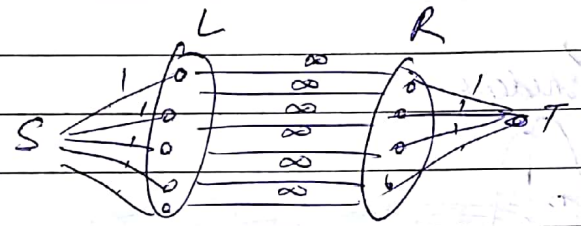
only for Bipartite graph, it is always true.



$|M| \leq |VC|$

vertex cover

Proof :-



$L_S = S \cap L$
$R_S = R \cap S$
$L_t = L \cap T$
$R_t = R \cap T$

$(u,v) \in E$
$u \notin L_T$, $u \in S$
$v \notin R_S$, $v \in T$.

not possible

$f(u,v) \leq c(u,v)$

define a cover, $C = L_T \cup R_S$

$\hookrightarrow$ min cut, max flow $\rightarrow$

$|M| = C(S,T)$
$= \sum C(s,u) + \sum C(v,t)$
$= |L_T| + |R_S|$

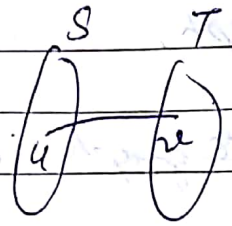
$(u,v) \in$ cut edge, $(u \in S, v \in T)$

$u \bullet \longrightarrow \bullet v$

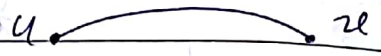
Can $s = u$?

$\hookrightarrow$ No!

I looks at all the edges reachable from S



$|M| = C(S,T) = \Sigma C(S,u) + \Sigma C(v,t)$



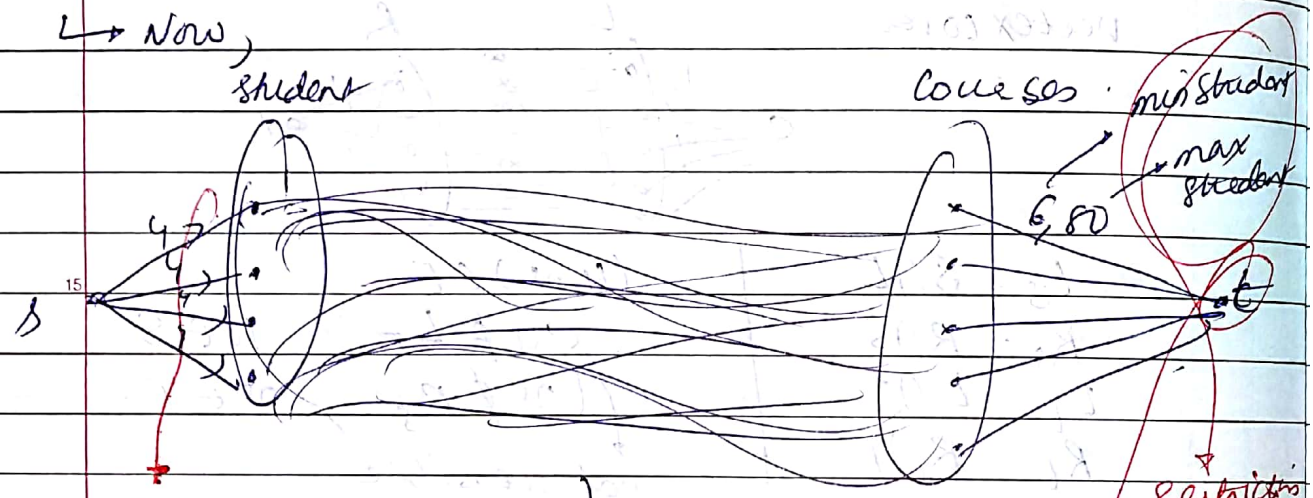
$u \in t$ or $v \in T$

$u \in S$ $v \in R$.

$|C| = |L_T| + |R_S|$

$\exists m$, st $|m| = |VC|$

↳ Now,



every student needs exactly 4 courses

restrictions

this is feasibility problem and not a maximization problem.

$1 \le d(s) \le 6$

at t
l, u

min # of students

max # of students

↳ ON RHS,

max → easy

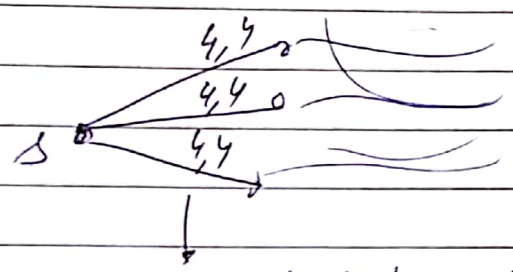
min → difficult.

lower bounds :-

capacity $c : E \to Z^+$
lower bound $l : E \to Z^+$ } feasibility

(1) $l(u,v) \le f(u,v) \le c(u,v) \ \forall (u,v) \in E$
(2) $f_{in} = f_{out} \quad \forall (u,v) \in E$

↳ for exactly 4- subjects



set both lowerbound & upperbound as '4' (min & max).

Now, convert it to the problem that has no lower bound & solve.