* Running time

1. $\log ab = \log a + \log b$.

2. $\log \frac{a}{b} = \log a - \log b$.

3. $\log a^b = b \log a$.

4. $a^{\log_b c} = b^{\log_c a}$.

5. $a^b = n$ then $b = \log_a n$.

22/09/22

Design and Analysis of Algorithms

Faculty: Salma Elmalaki (EH 3428) 11:15 – 12:15pm (Tue)
TA: Mojtaba Taherisadr (EH 3404) 9am – 10am (Fri)

* Algorithms
It is a finite, definite, effective procedure with some input and some output.

* wide useage

* Study of algorithm dates back to Euclid.
Formalized by Church & Turing

Q* What approach do we use
1) start from a broad problem (what)
2) formulate a problem precisely
3) Design an algorithm to solve it.
4) analyse the correctiveness
5) iterate if needed.

→ Chapter 1 : Introduction

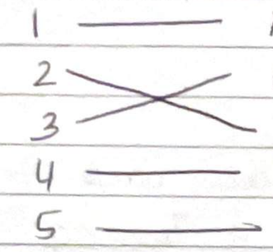
→ Stable matching.

I/p : 2 sets of elements & their preferences.

goal : Match elements from 2 sets

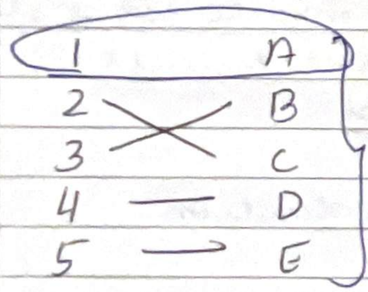
high level goal design a matching algo that converges, is distributed and self reinforcing.

* Match:
1 element is paired with an ∈ from the other set.

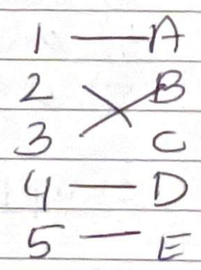


These ∈ from the 2 sets are matched.

* Perfect match:
a match is compute and no ∈ is left behind



Not perfect



* stable perfect matching.
no 2 elements such as (A, B) would want to form a pair over their current partners.

M ——— W M prefers W to W'
M' ——— W' M' " "
W' " M to M'
W " M to M'

* Problem: Stable Marrage

(1) Does a perfect stable matching always exists in stable marrage ⇐⇒ YES
~~(High/Highest) Not obvious~~

(2) Design an algo that finds a perfect stable match (Gale-Shapley algo)

(3) Analyse the algo
→ correctness ✓
→ uniqueness.
→ complexity.

(4) extension to other various problems.

* Formulating the problem requirements
- Perfect matching
- Stability
no incentive for some pair of participants to undermine assignment by joint action

* algo is correct if,-
1) it terminates
2) finds perfect matching
3) is stable

~~Proof of correctness~~

	1st	2nd	3rd	4th	5th.
victor	A	B	C	D	E
Wyatt	B	C	D	A	E
Xavier	C	D	A	B	E
Yancy	D	A	B	C	E
Zeus	A	B	C	D	E.

Amy	W	X	Y	Z	V
Bertha	X	Y	Z	V	W
Clare	Y	Z	V	W	X
Diane	Z	V	W	X	Y
Erika	V	W	X	Y	Z

Pseudocode

initialize each person to be free.

While (some man is free & hasn't proposed to every women)
{
  Choose such a man m
  W = 1st women on m's list, to whom he has not proposed

  If (w is free)
    m & w become engaged

  else if (w prefers m to her fiance m')
    m & w become engaged
    m' becomes free

  else
    w rejects m /* m remains free */
}

Best → n
Worst → n^2

* Proof of correctness

(1) Termination

claim: algo terminates after n^2 iteration

Obs 1: M proposes to w in ↓ order of preference
Obs 2: w can accept or trade.

Pf: each time m proposes to new women
there are only n^2 possible proposall

(2) Perfect matching

claim: all men & women don't get matched
Proof by contradiction
~P is false instead of P is true

Obs 1 & 2 → Same.

Pf: if Zeus is not matched upon algo termination
then some women is not matched (say Amy)
By obs 2 : Amy was never proposed to
but Zeus proposed to everyone

9/29/2022.

(3) Stability

claim algo does not return unstable pairs.

Pf: Suppose A-Z are unstable

Case 1: Z never proposed to A.
Z prefers GS partner over A.
A-Z Stable.

Case 2: Z proposes, A rejects.
A prefers GS partner over Z.
A-Z stable.

Gale-Shapley
finds a stable matching in at most n^2 iteration.

* Stable marriage problem always exists

* Stable roomate problem
a stable matching might not always exists.

2n people: each person ranks from 1 to 2n-1.

{GS assigns to each man his best valid partner}

Best valid is not his first preference.

M is valid partner for W if there exists stable matching in which they are matched

* All executions of GS yield man optimal assignment, which is a stable matching.

* Therefore GS is unique & man optimal.

(W,W') m — W (m,m')
(W,W') m' — W' (m,m')

Q. no of inputs
$n = 2$

Q. Size of the input
$N = 8$
(m, m', w, w', Pm, Pm', Pw, Pw')

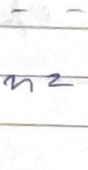
Q. How many combination of inputs are possible?
$(2)^4$ → four ppl in total with pref
→ options for each person

Q. Best case running time
2
Worst case
4

Now for input > 2.

Q What is the input
n, men's preference, women's preference

Q Size of input.
$n^2 + n^2 = 2n^2$



Q How many input are there
$(n!)^{2n}$ n arrangements of n things.

Q Worst Case n^2 Best case n

Q how fast is brute force ?
$n!$ npr

→ Polynomial time.

Desirable scaling property.

* When input size doubles, the algorithm should only slow down by a constant 'C'.

$C > 0$ & $d > 0$
such that for every input size N
running time cN^d steps.

Efficient

* An algorithm is efficient if its worst case running time is polynomial.

* If worst case is not polynomial then the algorithm is not efficient

→ Running time

$\log_2 n < n < n\log_2 n < n^2 < n^3 < 1.5^n < 2^n < n!$

eg: Running time

T(N) increase from cN^d to $C(2N)^d$

$C(2^d N^d)$

$2^d C(N^d)$

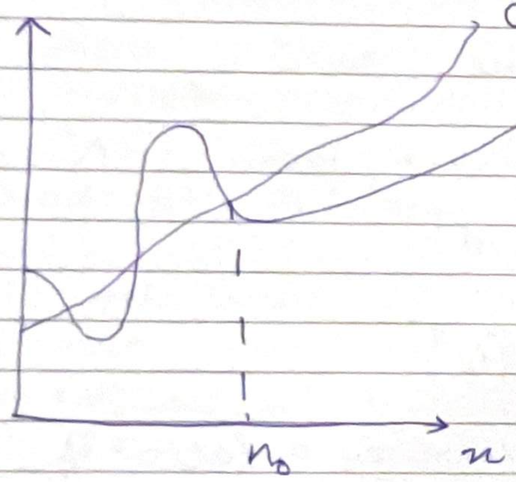
It slows down by a factor of '2^d'

→ Asymptotic Order of Growth

* Running time of algo depends on the function of 'N'

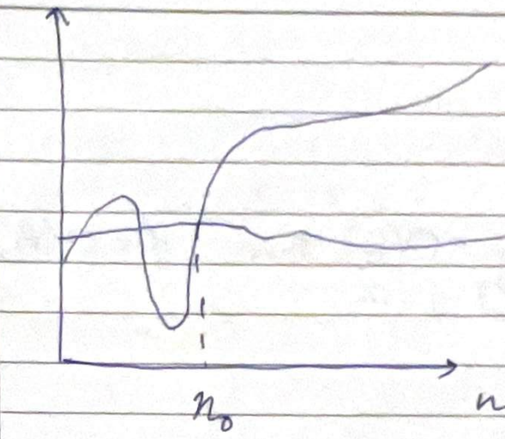
→ Upper bounds

$T(n)$ is $O(f(n))$ iff $c>0$ & $n_0 \geq 0$ such that all $n > n_0$ $\underset{n\to\infty}{}$ $T(n) \leq c \cdot f(n)$



→ Lower bounds

$T(n)$ is $\Omega(f(n))$ iff $c>0$ & $n_0 \geq 0$ such that all $n > n_0$ $T(n) \geq c \cdot f(n)$



→ Tight bounds

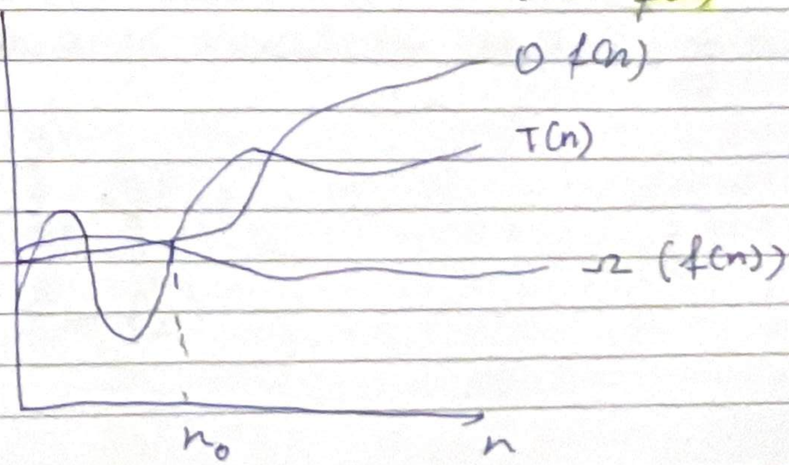
$T(n)$ is $\theta(f(n))$ iff $T(n)$ is both bound by $O(f(n))$ & $\Omega(f(n))$

if $\lim_{n\to\infty} \frac{T(n)}{f(n)} = c > 0$, then $T(n) = \theta(f(n))$

→ Tilda approx.

$T(n)$ $(\sim f(n))$ iff $\lim_{n\to\infty} \frac{T(n)}{f(n)} = 1$

Tight bounds



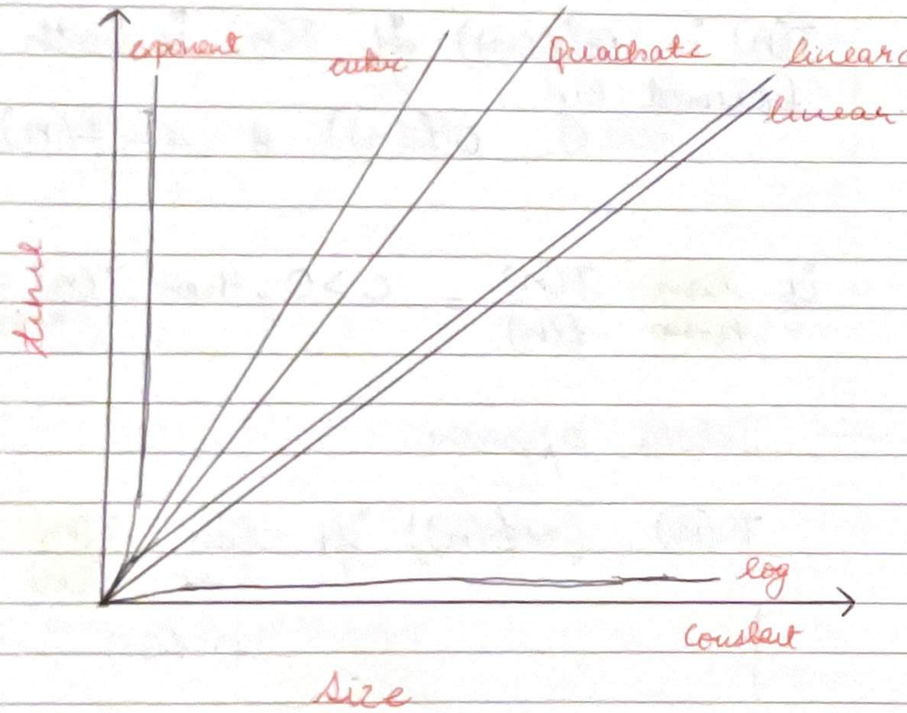
Properties.

1) Transitivity.

$f = O(g)$ & $g = O(h)$ then $f = O(h)$.
same for Θ & Ω.

2) Additivity.

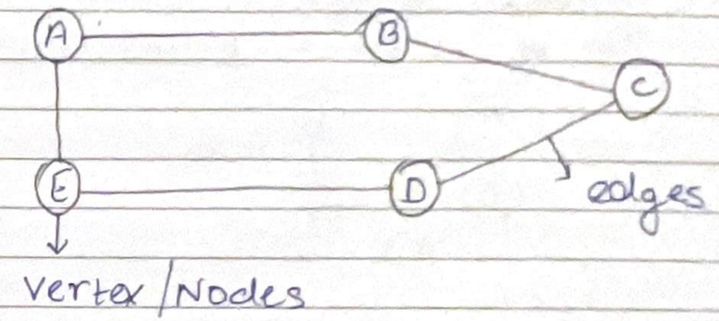
$f = O(h)$ & $g = O(h)$ $\quad f+g = O(h)$.



Graphs

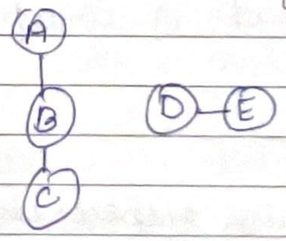
* non - linear data structure

$G = (V, E)$

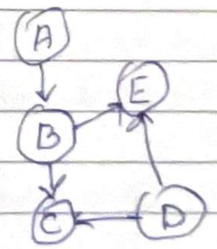


* Connected
There is a path from any vertices.

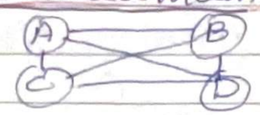
* disconnected.
There is no path between a node to another specific node.



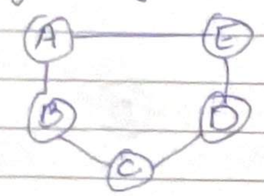
* directed graphs.
The edge depicts the direction of travel.



* Dense graph.
no of edges is close to maximum no of edges.



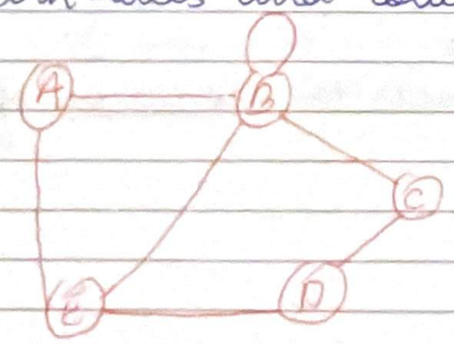
* Sparse graph.
No of edges = minimum no of edges.



* Degree Neighbour Indeg Outdeg.

No of ~~edges~~ vertices adjacent to a particular vertex.

→ 2 Ways to represent a graph
Adjacency matrix
A[v][v]
{2D array
Both rows and column represent vertices



	A	B	C	D	E
A	0	1	0	0	1
B	1	1	1	1	1
C	0	1	0	1	0
D	0	0	1	0	1
E	1	1	0	1	0

Advantages.
* adding/Removing an edge → O(1)
* T(n) to find adjacent vertices → O(1)

Disadvantages.
* consumes a lot of space. → $O(n^2)$
* Identifying all edges → $O(n^2)$
→ Adjacency list.

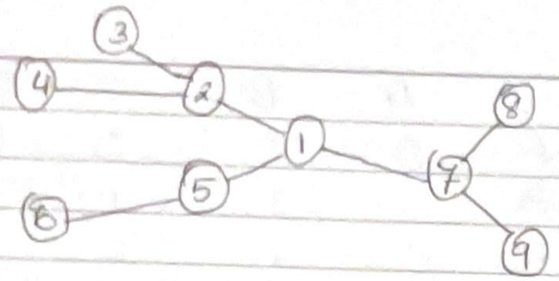
A → B → E
B → B → A → C → D → E
C → B → D
D → B → C → E
E → A → B → D

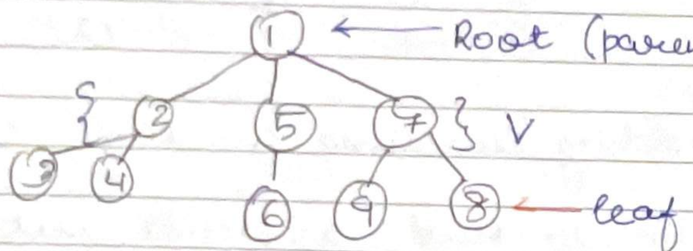
Advantage.
* space O(|V|+|E|) O[|V|+|E|]
* checking if (u,v) is an edge → O(deg(u)) O(log(u))

Tree

* An undirected graph is a tree if it is connected and does not contain a cycle

Tree = graph (if there exists no cycle)



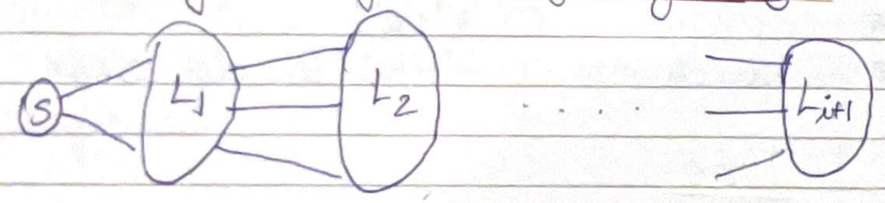


connected
undirected
NO cycle
} Theorem for tree

→ Graph traversal

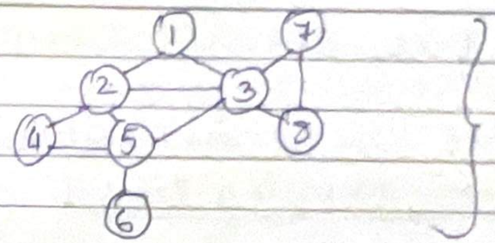
1) BFS
Breadth first search.

* exploring the graph layer by layer.



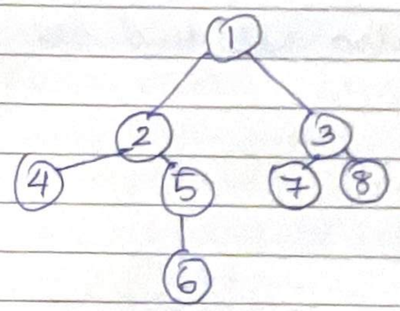
explore all adjacent node first.

IMP



graph A

BFS of graph A

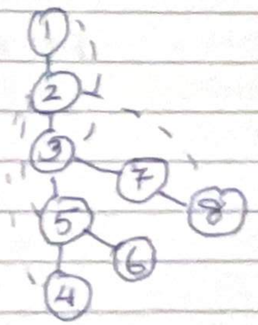


After BFS, the graph forms a tree.

* if an edge (x,y) exists in both the graph and tree the distance between x & y is 1

Depth first search

* explore each node as deep as possible, till leaf node is reach, then & backtrack to other parents to compute other branches.



India
India

* BFS & DFS trees are not unique
* BFS and DFS can be implemented to have O(m+n) running time.
* BFS finds path between s and t.
* BFS can also help find all connected components

CRYSTALS

* Solid matter exists in crystaline or ~~amphor~~ amorphus.

Crystaline → atoms are spatially arranged in a periodic fashion.

→ Single crystal

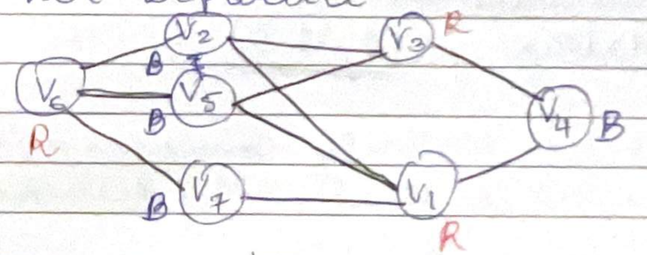
→ polycrystal

→ Amorphous.
no long term periodic structure.

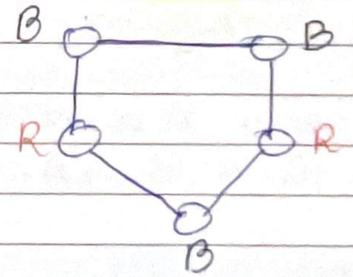
Bipartite Graphs

* An undirected $G = (V, E)$ is bipartite if the node can be coloured red or blue such that each edge has 1blue & 1red node.

How to test for bipartite
* explore in usual BFS way
  - at each node colour all its edges the opposite colour
  - If an edge has 2 nodes of same color it is not bipartite.



IMP * If a graph 'G' is bipartite then it can not have an odd length cycle.



* If a graph 'G' has 2 nodes connected in the same layer then it is not bipartite

If 'G' does not have 2 nodes connected in the ~~layer~~ same layer then graph is bipartite.

IMP Q. 'G' is bipartite and bipartitions of 'G' are 'U' and 'V'. What is the relation b/w them

Ans:

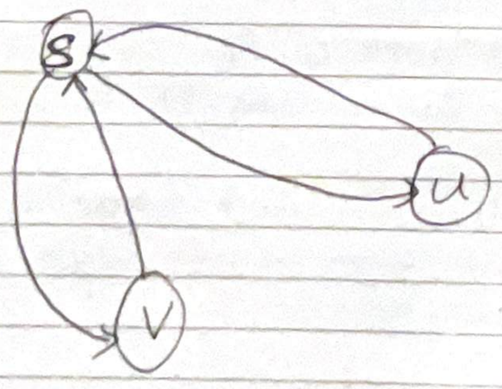
Sum of degree of vertices in U = Sum of degree of vertices in V.

Directed Graphs

* G = (V, E)
  edges goes from 'u' to 'v' in edge (u,v)
* Representation.
  Matrix : (1, -1, 0).
* Graph is strongly connected if every pair of nodes is mutually reachable

IMP → G is strongly connected iff every node is reachable from s and s is reachable from every node.

A iff B, first prove A and then B
Then you prove if B then A.



Path from u to v:
* Concatenate u-s path with s-v path.

path from v to u
* Concatenate v-s path with s-u path

Strong connectivity:

* Pick any node s
* Run BFS from s to every node
* Run BFS from every node to s

Return true iff all nodes reached in both BFS execution.

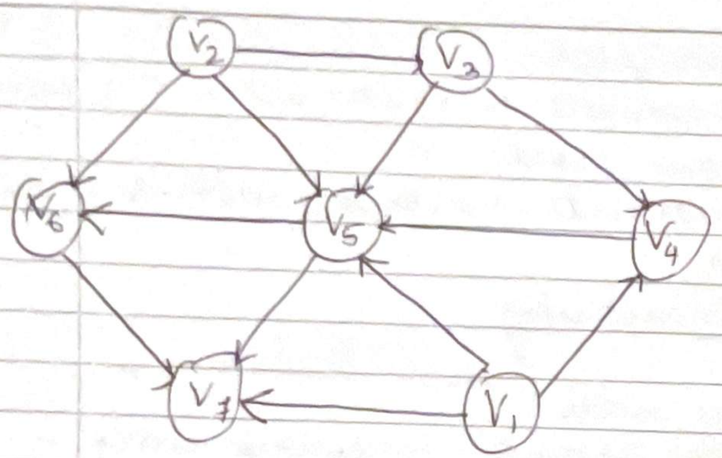
→ Directed Acyclic Graph

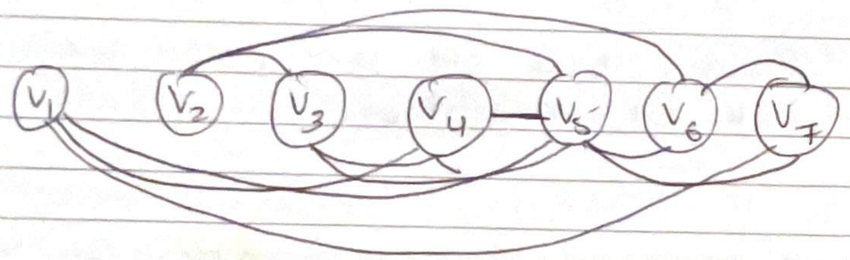
(DAG)

* A DAG is a directed graph that contains no directed cycle

like trees (tree = undirected graph without cycles)

A topological order of a direct graph $G = (V, E)$ is an ordering of its nodes as $v_1, v_2, v_3 \dots v_n$ so that all edges (v_i, v_j) point forward $(i < j)$.





If 'G' is a topological order then 'G' is a DAG.

* find a node with no incoming edges

Q If G is a DAG, then G has a topological order.

proof by induction.

Proof by induction (Third proof)

* Prove something for all natural numbers

(1) first proving that it is true for 1.

IH (2) if it is true for 'n' inductive hypothesis
(or atleast all numbers upto n)

(3) then it is also true for n+1

n = 1 has a topological order

If G is a DAG, then G has a topological ordering.

* if n=1, because topological ordering is G

* If G ~~has~~ is DAG of size < n, then G has topological ordering

* given a DAG G with n+1 nodes

find v with no incoming edges

G - {v} is a DAG
since deleting v does not create cycles

By IH G - {v} has a topological ordering
place v first, then append topological ordering of G - {v}

This is valid since v has no incoming edges.

Running time = $O(m+n)$.

☆ Topological ordering of a graph is not unique

Greedy Algorithm

☆ An algorithm that locally optimizes some measure at every step, to find a global solution.

☆ Are not always optimal.

→ Feasible solution: Solution that satisfy our constraints are called feasible.

→ Optimal solution

A soln which is feasible and that is giving me the best result.

- There can be only 1 optimal soln.

- Greedy is not always optimal
- They are optimal when the problem has some structure.

→ How to prove greedy is optimal

1) The greedy stays ahead.
The progress made by greedy at any step is greater than any other algorithm.

2) exchange argument.

Optimal solution is converted into a greedy soln without effecting its quality.

3) Structural bound.

every soln has some value and it reaches that bound.

→ Interval Scheduling

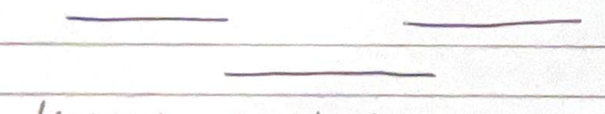
Task: find job that dont overlap and find maximum subset of mutually compatible jobs.

Template

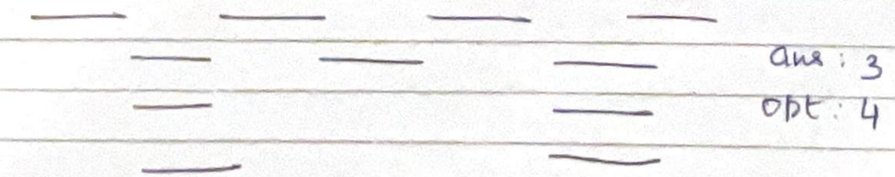
1) earliest start time : ascending order of start time.
2) earliest finish time: ascending order of finish time.
3) Short interval : which have the shortest interval.
4) fewest conflicts : less overlap.

early start:

short interval.



fewest conflict.



Theorem Greedy is optimal

$G = \{ g_1, \dots g_k \}$

$H = \{ h_1, \dots, h_l \}$

we want $k = l$

Lemma $i \leq k$ $\quad f(g_i) \leq f(h_i)$

→ Base Case: $i = 1$

~~IH~~ greedy choose g_1 which has the earliest finish time

Thus $f(g_1) \leq f(h_1)$.

→ IH.

$f(g_i) \leq f(h_i) \Rightarrow f(g_{i+1}) \leq f(h_{i+1})$

→ greedy will find a task that is

$R_j = (s_j, f_j)$ $\quad f_j$ is smallest

where $s_j \geq f_{g_i}$

$f(g_i) \leq f h_i \leq s(h_{i+1}) < f(h_{i+1})$.

IH — since H is compatible.

Thus greedy has l_{i+1} as an option but the finish time could be before that.

Thus $f(g_{i+1}) \leq f(l_{i+1})$.

Running time

$O(n \log n)$

sorting → $O(n \log n)$.

→ Interval Partitioning.

* lower bound on optimal soln.

* The depth of a set of open interval is the maximum number that contain any given time.

i.e No. of overlapping requests.

Here we use start time of a task

Greedy Analysis

fact 1: Greedy assigns every interval I_j some label out of d total.

fact 2: Greedy never gives 2 overlapping lectures the same label.

Depth: The maximum no of overlapping tasks.

→ Scheduling to minimize lateness.

Job 'J' needs 't_j' unit of processing time and is due at time 'd_j'

If it starts at time s_j, it finishes at time $f_j = s_j + t_j$

Lateness $l_j = \max\{0, f_j - d_j\}$.

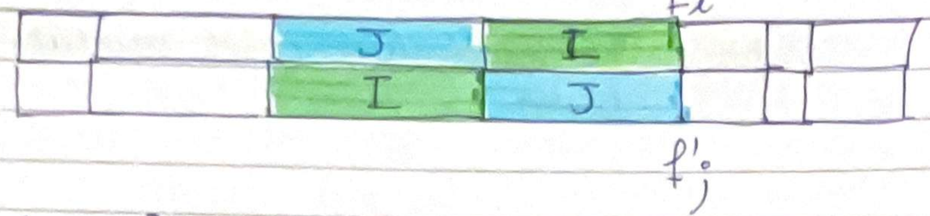
Goal: to minimise max lateness.

* There exists an optimal schedule with no ~~optimal~~ idle time.

INVERSION

* Such a pair where $i < j$ i.e $(d_i \leq d_j)$ but j is scheduled first is called Inversion.

* greedy has NO inversions



Swapping: enhances inversion by 1 & does not increases max lateness.

$$l'_k = l_k \text{ for all } k \neq i, j$$

$$l'_i \leq l_i$$

$$l'_j = f'_j - d_j$$

$$= f_i - d_j \quad (j \text{ finishes at } f_i)$$

$$\leq f_i - d_i \quad (i < j)$$

$$\boxed{l'_j \leq l_i}$$

Asymptotic

Nature of function for a very large n

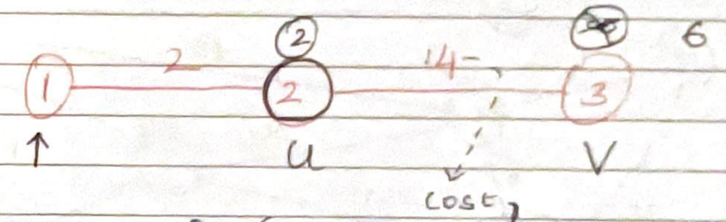
upper bound [O( )] Time & space

$T(n)$ is $O(f(n))$ iff $T(n) <= c*f(n), n > n_0, n_0 > 0$

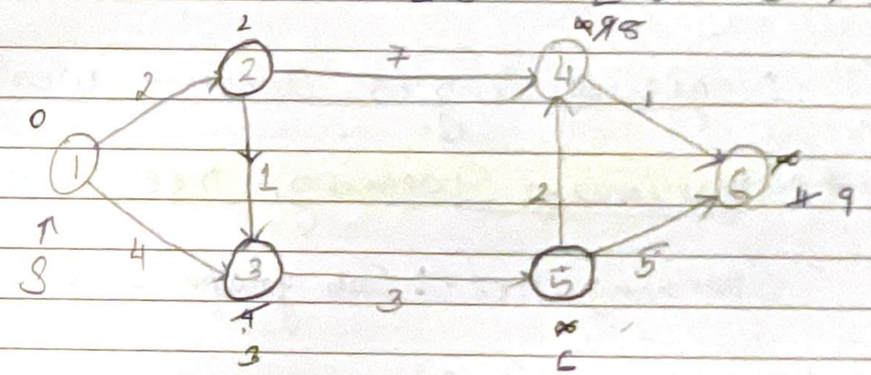
Shortest Path

* Directed graph. $G = (V, E)$
Source 's'
destination 't'
length of edges = l_e

find the shortest path from s to t.



* Relaxation: if $(d[u] + c(u,v) < d[v])$
then $d[v] = d[u] + c(u,v)$



v	dv
2	2
3	3
4	8
5	6
6	9

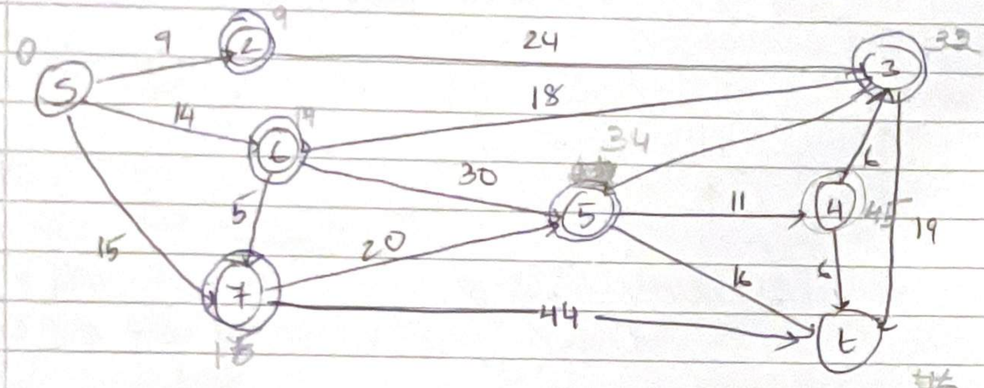
Running time :

no of vertices = n.

it may relax all vertices = n.

∴ Running time = $O(n^2)$.

with 'priority queue' it is $O(m \log n)$



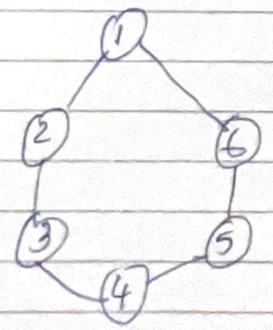
Negitive lengths doesn't work

→ Minimum Spanning tree

Spanning tree : Sub graph of a graph.

$G = (V, E)$

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



$E = \{(1,2), (2,3) \ldots (6,1)\}$

Spanning tree is a subset of graph where it is represented by the set of edges.

Number of spanning trees possible

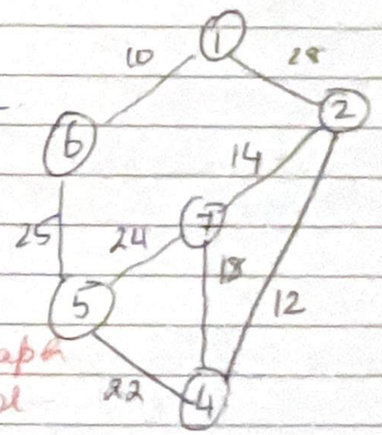
n^{n-2}

* How to find MST using greedy

1) Prims
2) Kruskals
3) Reverse - delete

Prims Algorithm

- Select a minimum cost edge
- Select a minimum cost edge with already connected vertices.

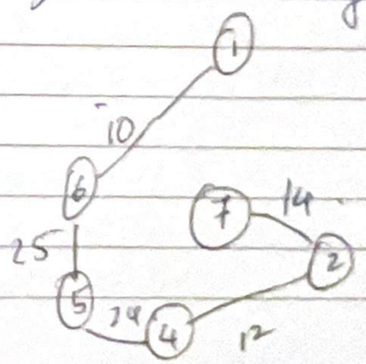


For non - connected graph MST can not be found

$O(n^2)$ → array

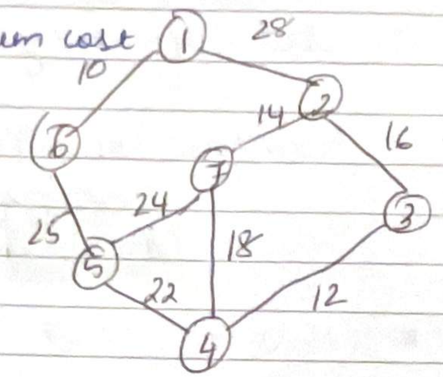
$O(m \log n)$ → binary heap.

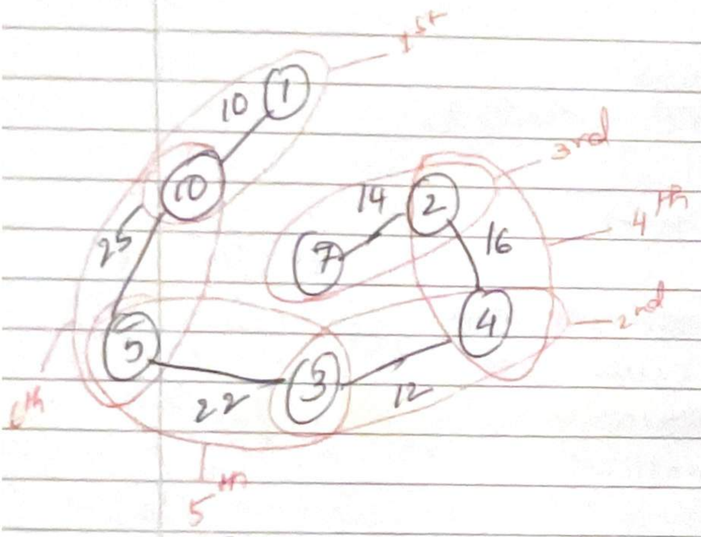
$|E|_G$ - No of MST cycles



Kruskals Algorithm

- Always select a minimum cost edge, but should not form a cycle



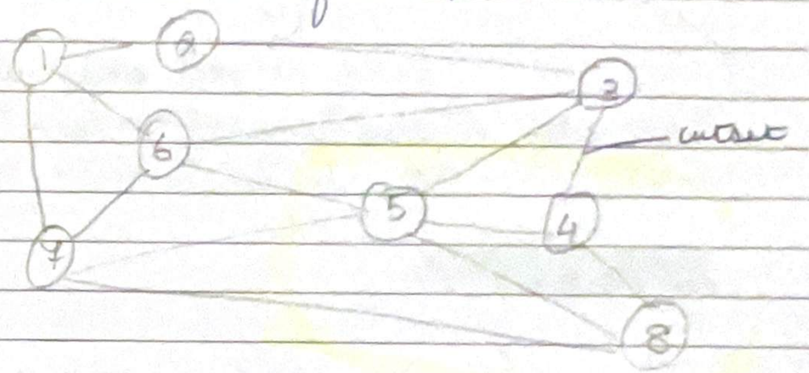


RT
$O(n^2)$ ⟶ array
$O(n \log n)$ ⟶ Min heap.

Cycle: Set of edges of the form

a-b, b-c, ..., z-a

Cutset: The set of edges (of a cut) whose endpoints are in different subsets of the partition



cut S: {4,5,8}

cutset D: {5-6, 5-7, 3-4, 3-5, 7-8}

A cycle and a cutset intersect in an even number of edges.

Cut property: S is any subset of nodes, and 'e' be the minimum cost edge with exactly 1 endpoint in S. The MST must contain e

Cycle property: Let C be any cycle, and f be the max cost edge belonging to C. Then the MST does not contain f

IMP

Cut property

proof by contradiction.

We prove that 'e' that did not belong to the spanning tree now belongs to ST
T* as

$C_e < C_{e'}$, $cost(T) < cost(T^*)$

e' edge with higher wgt.

We added 'e' and it creates a cycle. that's why

Cycle property

$T' = T^* \cup \{e\} - \{f\}$ is also a spanning tree (connected, no cycle)

Since $C_e < C_f$, $cost(T') < cost(T^*)$

∴ f can be removed from T^*

Reverse - Delete Algorithm

* Consider edges in descending order of cost & T = E

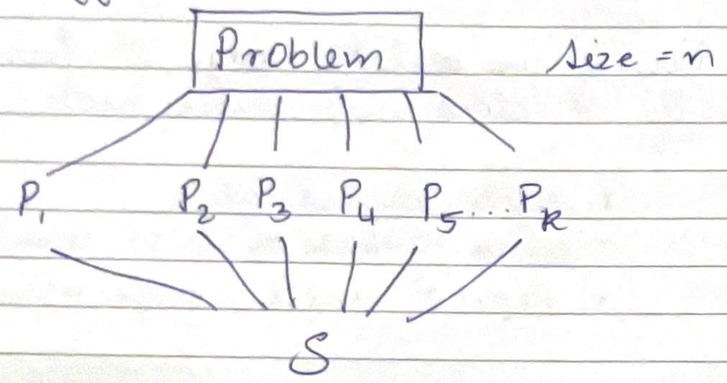
Now, if we delete edge 'e' from T, only if it does not disconnect.

* delete largest edge
* check graph is still connected
* Repeat until edge = vertices - 1

Chapter 5

Divide and Conquer.

* Stratergy to solve a problem



- if a sub problem is also large apply D & C.

- It is recursive in nature.
- The problem & subproblem should be of the same type.

→ Merge sort

Algorithm

```
Mergesort (l,h)
  if (l<h)
  {
    mid = (l+h)/2;
    Mergesort (l, mid);
    Mergesort (mid+1, h)
```

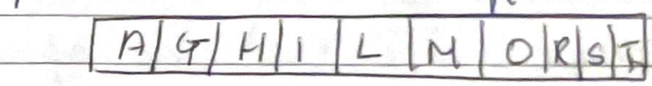
Merge Sort

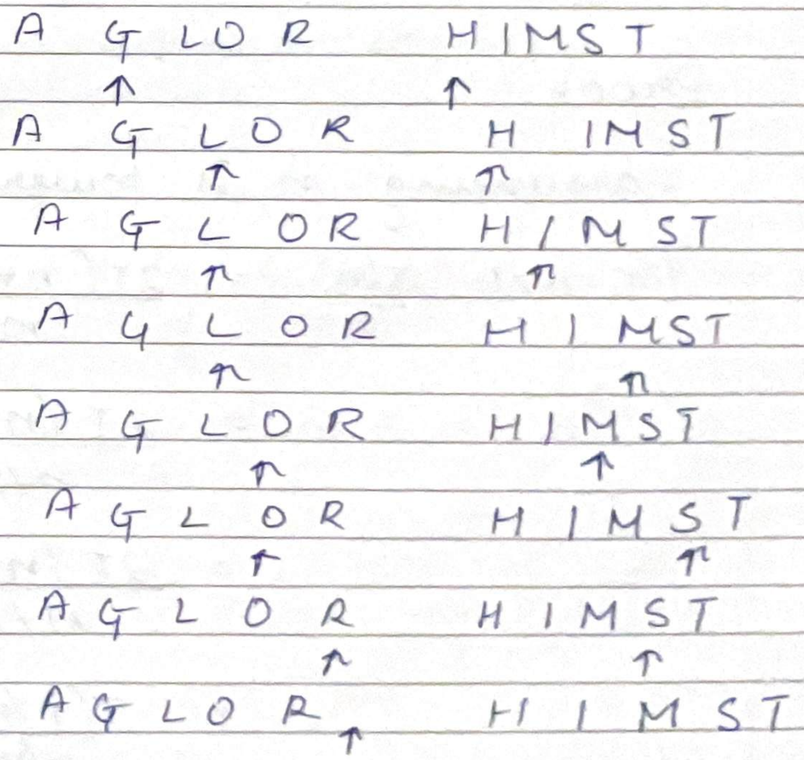
* Divide array into 2 halfs
* Recursively sort each half.
* Merge 2 halfs to make a sorted array whole.

A G L O R H I M S T

* how to merge.

A G L O R H I M S T.





Running time

mergesort recurrence.

$T(n)$ = no of comparisons to mergesort

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2T(n/2) + n & \text{otherwise} \end{cases}$$

soln: $O(n \log_2 n)$

merge — $\log_2$ → no of levels.

Proof

(1) assuming n is power of 2.

for $n > 1$ $\quad \frac{T(n)}{n} = \frac{2T(n+2)}{n} + 1$

$$= \frac{2T(n/2)}{n/2}$$

$$= \frac{2T(n/4)}{n/4} + 1 + 1$$

$$= \frac{T(n/n)}{n/n} + \underbrace{1 \ldots + 1}_{\log_2 n}$$

$$= \log_2 n$$

$T(n) = n \log n$

Q.

Divide into 'q' subproblem of size $n/2$. each and merge in $cn = O(n)$

$q = 1$ linear

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

$q = 2$ linear

$$T(n) \leq \begin{cases} c & \text{if } n = 2 \\ 2T(n/2) + cn & n > 2 \end{cases}$$

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

$q > 2$

$$T(n) \leq \begin{cases} c & \text{if } n = 2 \\ qT(n/2) + cn & n > 2 \end{cases}$$

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

Master theorem

if $T(n) = a\,T(n/b) + O(n^d)$

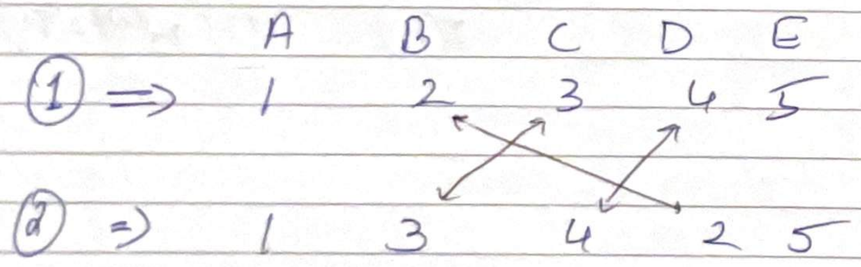
for $a > 0,\ b > 1,\ d \geq 0$

P.T

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

a = no of subproblem
b = size of each problem.
c = cost to do the merge.

→ Counting inversions



Inver

* Condition for inversion

i and j are inverted if $i < j$, but $a[i] > a[j]$

Q possible inversions

1, 3, 4, 2, 5.

Brute force → time complexity

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

(2 inversions)

divide and conquer

1 5 4 8 10 2 6 9 12 11 3 7.

divide O(1)

1, 5, 4, 8, 10, 2 | 6, 9, 12, 11 3, 7

conquer

2 T(n/2)

5-4, 5-2, 4-2, 8-2, 10-2 (5) | 6-3, 9-3, 9-7, 12-11, 12-3, 12-7, 11-3, 11-7 (8)

Count the inversion b/w the 2 arrays → (9)

Total = 5 + 8 + 9 = 22

Count this using merge sort.

Merge Count

* Step to count inversion b/w 2 sorted arrays.

eg. A: 3 7 10 14 18 19 ; B: 2 11 16 17 23 25

C=6

3 7 10 14 18 19 | ~~2~~ 11 16 17 23 25 (6)

C=6

~~3~~ 7 10 14 18 19 | ~~2~~ 11 16 17 23 25 (6)

C=5

~~3~~ ~~7~~ 10 14 18 19 | ~~2~~ 11 16 17 23 25 (6)

C=4

~~3~~ ~~7~~ ~~10~~ 14 18 19 | ~~2~~ 11 16 17 23 25 (6)

C=3

~~3~~ ~~7~~ ~~10~~ 14 18 19 | ~~2~~ ~~11~~ 16 17 23 25 (6)

C=3

~~3~~ ~~7~~ ~~10~~ ~~14~~ 18 19 | ~~2~~ ~~11~~ 16 17 23 25 (6 3)

C=2

~~3~~ ~~7~~ ~~10~~ ~~14~~ 18 19 | ~~2~~ ~~11~~ ~~16~~ 17 23 25 (6 3)

C=2

~~3~~ ~~7~~ ~~10~~ ~~14~~ ~~18~~ 19 | ~~2~~ ~~11~~ ~~16~~ ~~17~~ 23 25 (6 3 2)

C=2

~~3~~ ~~7~~ ~~10~~ ~~14~~ 18 19 | ~~2~~ ~~11~~ ~~16~~ ~~17~~ 23 25 (6 3 2 2)

~~3~~ ~~7~~ ~~10~~ ~~14~~ ~~18~~ ~~19~~ | ~~2~~ ~~11~~ ~~16~~ ~~17~~ ~~23~~ 25 (6 3 2 2)

~~3~~ ~~7~~ ~~10~~ ~~14~~ ~~18~~ ~~19~~ | ~~2~~ ~~11~~ ~~16~~ ~~17~~ ~~23~~ ~~25~~ (6 3 2 2)

No of inv = 6 + 3 + 2 + 2 = 13

2	3	7	10	11	14	16	17	18	19	23	25

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

Pre-Cond: A & B are sorted (Merge & count)

Post Cond: L is sorted (sort & count)

→ Integer Multiplication

- 2 n-digit integer a and b, compute a+b
  $\theta(n)$.
- Multiply 2 n-digit
  brute force $O(n^2)$

First Method:

2 n-digit number.
Multipy 4 1/2 n digit integer.
add 2 1/2 n integer & shift to get result

$x = 2^{n/2} \cdot x_1 + x_0$

$y = 2^{n/2} \cdot y_1 + y_0$

$xy = 2^n x_1 y_1 + 2^{n/2}(x_1 y_0 + x_0 y_1) + x_0 y_0$

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

Using masters algoriten we find that

$a = 4$

$b = 2$

$d = 1$

and $d < \log_b a$

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

So, now we try

Karatsuba Multiplication

* add 2 $1/2\ n$ digit integer
* Multiply 3 $1/2\ n$ digit integer
* add, Subtract and Shift $1/2\ n$ digit int.

$$x = 2^{n/2} x_1 + x_0$$
$$y = 2^{n/2} y_1 + y_0$$

$$xy = 2^n \cdot x_1y_1 + 2^{n/2}(x_1y_0 + x_0y_1) + x_0y_0$$

$$= 2^n \underbrace{(x_1y_1)}_{A} + 2^{n/2} \underbrace{(x_1+x_0)(y_1+y_0)}_{B} - \underbrace{x_1y_1}_{A} - \underbrace{x_0y_0}_{C} + \underbrace{x_0y_0}_{C}$$

$T(n) \leq 3T(n/2) + c(n)$

Using master theorem

$a = 3, b = 2, d = 1$

$d < \log_b a \Rightarrow O(n^{1.585})$

Matrix Multiplication

→ given 2 ~~$n/2$~~ $n \times n$ matrix A & B. compute AB.

Brute force = $O(n^3)$

* A & B into $1/2\ n \times \frac{1}{2} n$ blocks
* Multiply $8\ 1/2\ n \times 1/2\ n$ recursively.
* add using 4 matrix addition.

$$T(n) \leq 8T(n/2) + \Theta(n^2)$$
$$\Rightarrow T(n) = \underline{\Theta(n^3)}$$

* Make it 7 multiplication

~~$T(n) \leq 7T(1)$~~

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

$$\Rightarrow T(n) = \Theta(n^{\log_2 7}) = \underline{O(n^{2.81})}$$

Best know $O(n^{2.376})$

Coppersmith – Winograd.

Dynamic programming

* Break up a problem into series of overlapping sub problem and build up solutions to larger and larger sub-problems.

Here subproblems are reused several times.

→ Weighted interval scheduling.

* Job j starts at s_j finishes at f_j has value v_j
* 2 jobs are compatible if they dont overlap.
* find max weighted subset of mutually compatible jobs

P(j) = largest index $i < j$ such that 'previous' Job i is compatible with j

* Start with last Job.

OPT(J) = Value of optimal sol to the problem consisting of Job request 1, 2, ..., j

Case1: OPT selects J.
no compatible jobs $\{P(j)+1, P(J)+2, \ldots, j-1\}$

Must include opt sol to prob consisting of remaining compatible job 1, 2, ..., P(j)

Case 2: OPT does not select job J.

Opt soln 1, 2, ..., j-1

Bellmans eq

$$OPT(J) = \begin{cases} 0 & \text{if } J=0 \\ \max\{v_j + OPT(P(j)), OPT(j-1)\} & \text{otherwise} \end{cases}$$

Req j belongs to an optimal sol iff

$v_j + OPT(P(J)) > OPT(j-1)$

Recursive : has redundant sub problem

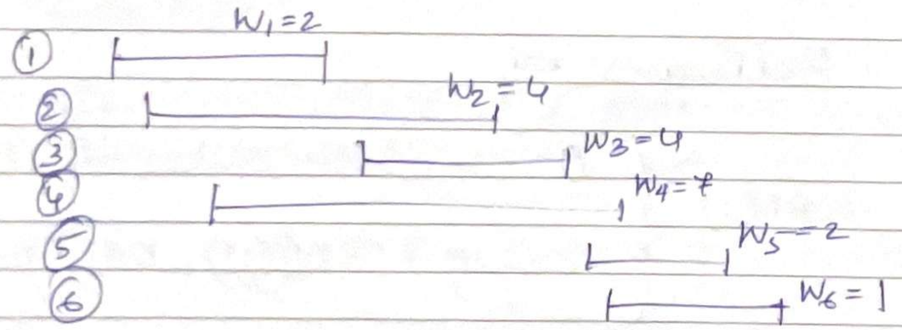
Memoization

→ store results of each sub problem in a cache

storing in global array.

Running time : $O(n)$

→ Bottom-up dynamic programming
↳ unwind recursion



Job	1	2	3	4	5	6
P(J)	0	0	1	0	3	3
OPT()						

To calculate OPT() use the eq:
$w_J + OPT(P(J))$, $OPT(J-1)$

Job	1	2	3	4	5	6
P(J)	0	0	1	0	3	3
OPT()	2	4	6	7	8	8

∴ OPT is 8 with Job 5, 3, 1

→ Knapsack Problem

* optimise the problem.
* Sequence of decisions
* try all possible solution and pick up the best one.

* Total sol 2^n $T(n) = O(2^n)$ time consuming.
* goal to max value

$m = 8$, $n = 4$

a) Tabulation

P	w_i	weights → i	0	1	2	3	4	5	6	7	8
		0	0	0	0	0	0	0	0	0	0
1	2	1	0	0	1	1	1	1	1	1	1
2	3	2	0	0	1	2	2	3	3	3	3
5	4	3	0	0	1	2	5	5	6	7	7
6	5	4	0	0	1	2	5	6	6	7	8

$V[i,w] = \max\{V[i-1,w],\ V[i-1, w-W[i]] + P[i]\}$

$V[4,1] = \{V[3,1],\ V[3,1-5] + 6\}$

$V[4,5] = \max\{V[3,5],\ V[3,5-5] + 6\}$

$5 \qquad 0+6 = 6$

$V[4,7] = \max\{V[3,7],\ V[3,7-5] + 6\}$

$7, \quad 1+6 = 7$

Profit = Total(8) − P(4) = 2

	x_1	x_2	x_3	x_4
	0	1	0	1

* 8 is only in the last row, which means that 4th object should be included in the optimised problem.

Profit = Total value (8) − P(4) = 2.

The remaining profit we need is 2 so, we check 3rd object, but is present before that two, so we dont include third object.

2 is not there before ~~seven~~ 2nd obj so we take second obj into the optimal sol.

$W = 8$

∴ x_2 & x_4 are used

obj	1	2	3	4
P =	1	2	5	6
W =	2	3	4	5

(b) Sets method.

* a set has the elements Profit & weight.

$S^0 = \{0,0\}$

$S_1^0 \{(1,2)\}$ → obj 1.

$S^1 = \{(0,0), (1,2)\}$

$S_1^1 = \{(2,3)(3,5)\}$ exceeding the weight.

$S^2 = \{(0,0), (1,2), (2,3), (3,5)\}$

$S_1^2 = \{(5,4), (6,6), (7,7), (8,9)\}$

$S^3 = \{(0,0), (1,2), (2,3), (5,4), (6,6), (7,7), (8,9)\}$

here weight increased so we discard (3,5)

This is dominance rule.

$S_1^3 = \{(4,5), (7,7), (8,8), (11,9), (12,11), (13,12)\}$

$S^4 = \{(0,0), (1,2), (2,3), (5,4), (7,7), (6,5), (8,8)\}$

Our highest set is (8,8)
We use this to solve

$(8,8) \in S^4$
but $(8,8) \notin S^3$

$\therefore x_4 = 1$

$(8-6, 8-5)$ Sub the profit & weight of the fourth obj.
↓
$(2,3)$

$(2,3) \in S^3$
$(2,3) \in S^2$
but $\notin S^1$

$\therefore x_2 = 1$

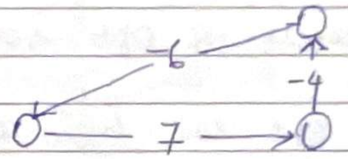
$\therefore$ obj 2 & 4 form the opt sol.

$$OPT(i,W) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, W) & \text{if } w_i > W \\ \max\{OPT(i-1, W), v_i + OPT(i-1, W-w_i)\} & \text{Otherwise} \end{cases}$$

Shortest path using Dynamic Programming

* Shortest path with negitive weights

→ Negitive cost cycle

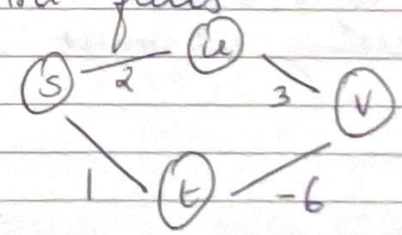


* There is no shortest path if there is a negitive cost cycle

if there is no -ve cycle

then → exists a shortest path V-t path & it is a simple path
↳ can not repeat
↳ length = n-1

dijkstra fails



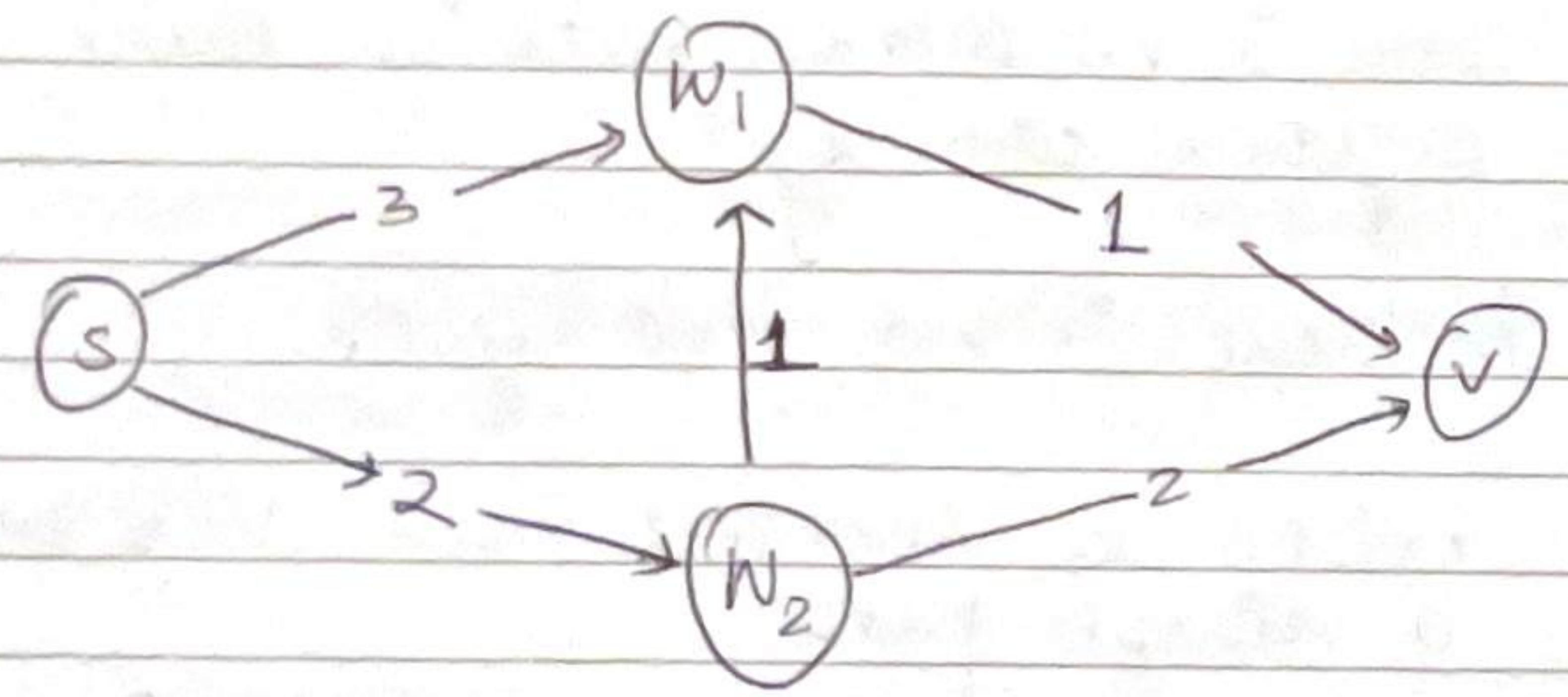
dijkstra - s-u, u-v
= 5
opt - s-t, t-v
= -5

__/__/__

Recipe

- define subproblem
- Recursively define Value of opt sol
- compute Value of opt sol
- construct opt sol from computed informatio

Q.



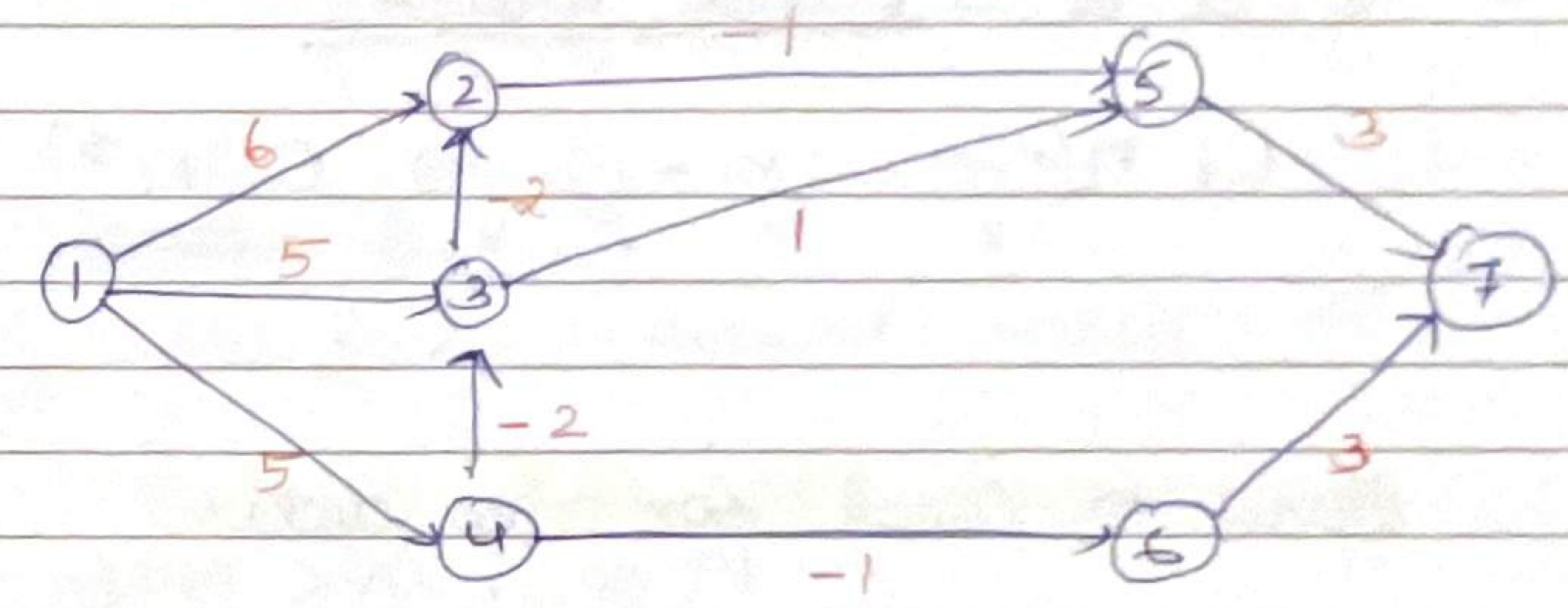
through which edge (W,V) do we get shortest path with atmost 'i' edges.

if we pick 'W', what is the shortest path to W using at most i-1 edge

__/__/__

Single source shortest path (Bell-ford)

Relaxation - if $(d(u) + C(u,v) < d(v))$
then $d(v) = d(u) + C(u,v)$

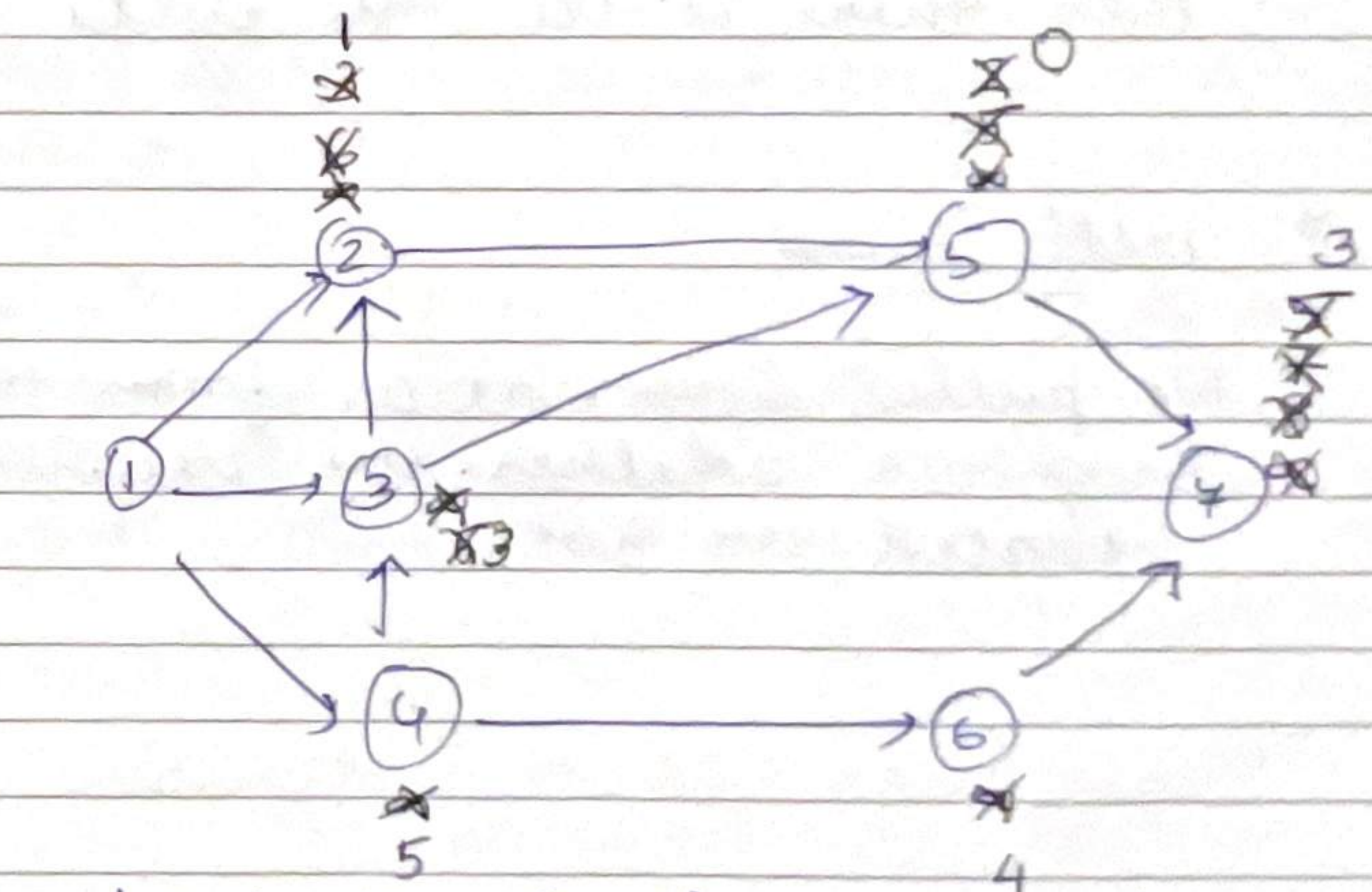


edge list

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

iteration
1 ✓
2 ✓
3 ✓
4 ✓
5

no change so stop



do n-1 iterations to get shortest path to each node.

1 → 0	4 → 5	7 → 3
2 → 1	5 → 0	
3 → 3	6 → 4	

No Negitive cycle

$OPT(i,v) = OPT(n-1,v)$

Running time

$O(|E| \times |V|-1) \Rightarrow O(|V||E|) \Rightarrow O(n^2) \Rightarrow O(n^2)$

* if it is a dense graph.

$O\left(\frac{n(n-1)}{2}(n-1)\right) \Rightarrow O(n^3)$

How to Check for -ve cycle.

$OPT(n,v) < OPT(n-1,v)$

if suppose you do 'n' iteration & relaxation of an edge occurs then there is a -ve cycle.

* pull based

We pulled information from the neighbors whether the neighbor are updated or not.

stop algo at n-1 iteration

if no change before n-1 iteration stop then.

$$OPT(i,v) = \begin{cases} \infty & \text{if } i = 0 \\ \min\left\{OPT(i-1,v), \min\{OPT(i-1,w) + c_w\}\right\} \end{cases}$$

→ how to impove.

1) check only changed nodes
2) $OPT(v) \leftarrow \min\{OPT(v), OPT(N) + c_N\}$

→ Detecting negitive cycles

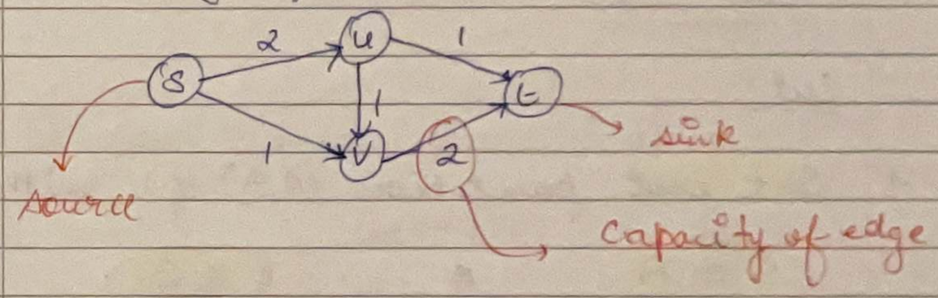
* add a node t, connect all nodes to t with cost 0.

* run BF, check if $OPT(n,v) = OPT(n-1,v)$ for all nodes going towards t.
if true, no neg cycle
else, there is a neg cycle

7 ⟶ Network flow & cuts Chapter 7

* Directed Graph
  $G = (V, E)$

* 2 important nodes 's' & 't'
* capacity of e^{th} edge = $C(e)$



* Constraint

1) Capacity : for each $e \in E$
   $0 \le f(e) \le C(e)$

2) Conservation : for each $v \in V - \{s, t\}$
   $\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e)$

* Value of flow :
   $v(f) = \sum_{e \text{ out of } s} f(e)$.

* Max flow

* S-t flow has max value.

$$v(f) = \sum_{e \text{ out of } s} f(e)$$

* Min cut

Cut

* S-t cut partition (A,B) of V with

$s \in A$ & $t \in B$

* $Cap(A,B) = \sum_{e \text{ out of } A} c(e)$

* Find an S-t cut with minimum capacity

Flow value

f - any flow
(A,B) - any S-t cut

Then flow ~~value~~ sent across the cut is equal to the amount leaving s.

* $v(f) \leq cap(A,B)$ Weak Duality

* Max flow - Min cut theorem

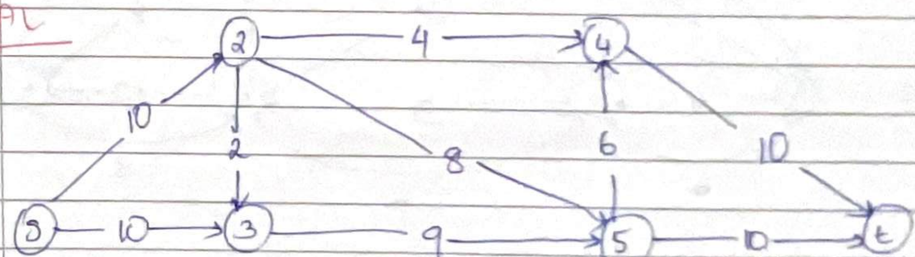
$$v(f) = cap(A,B)$$

f = Max flow
A,B = Min cut

Residual Graph.

$$c_f(e) = \begin{cases} c(e) - f(e) & \text{if } e \in E \quad (e = (u,v)) \\ f(e) & \text{if } e^R \in E \\ & \hookrightarrow e^R = (v,u) \end{cases}$$

ORIGINAL



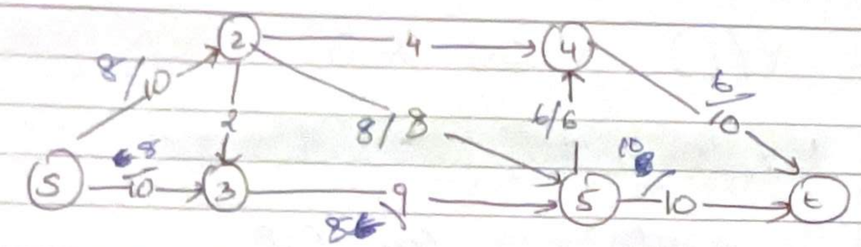
find a path, compute bottleneck, augment each edge and total flow.

* DFS used to find path.

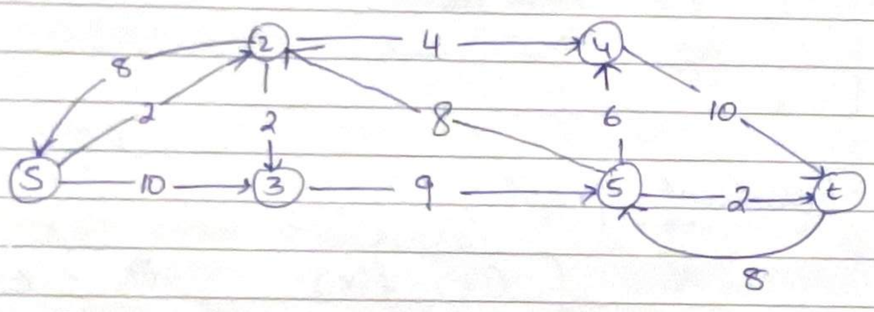
* Run time

$$= O(Ef)$$

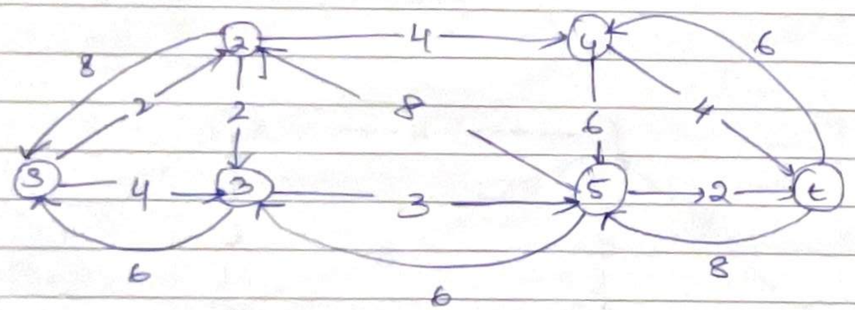
no of edge f - max flow



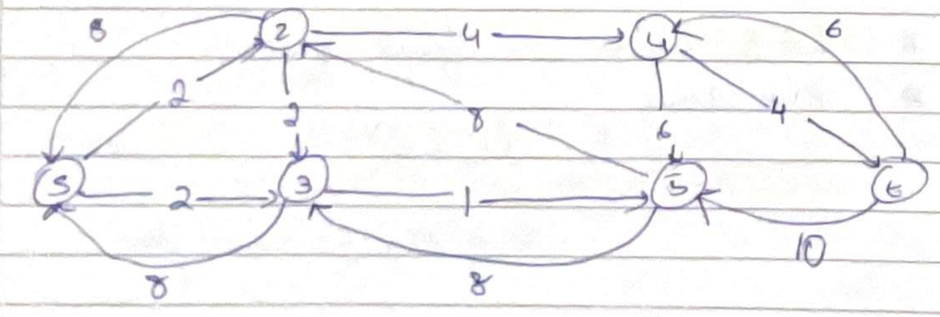
(1) S - 2 - 5 - t BN = 8



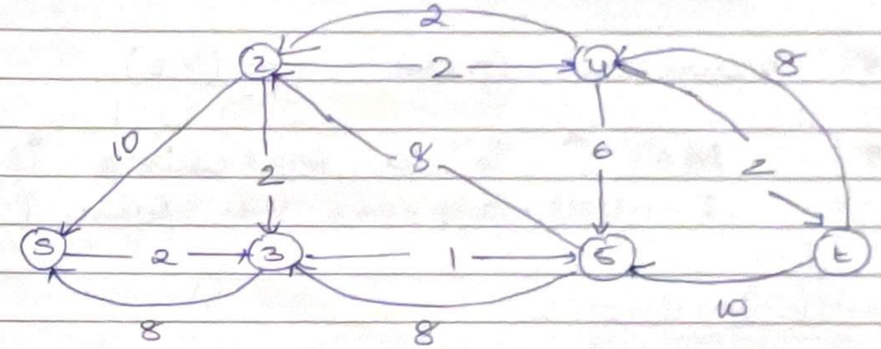
(2) S - 3 - 5 - 4 - 10 . BN = 6



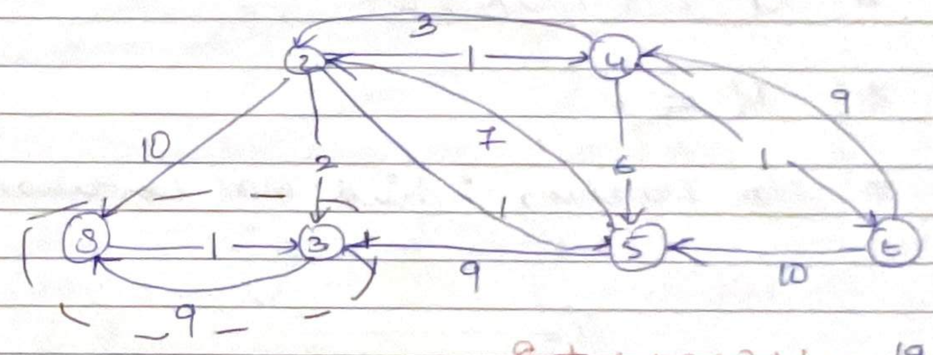
(3) S - 3 - 5 - t BN = 2



(4) S - 2 - 4 - t BN - 2



(5) S - 3 - 5 - t. BN = 1



8 + 6 + 2 + 2 + 1 = 19

* Here termination is guaranteed if capacities are integers

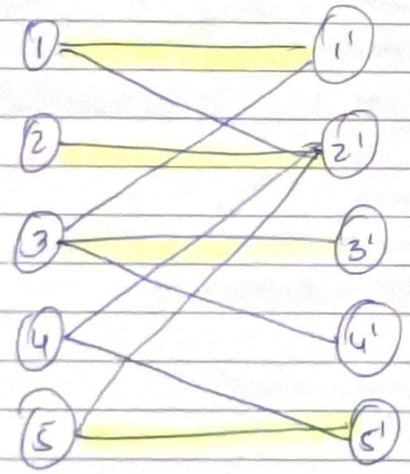
Bipartite Matching

Matching

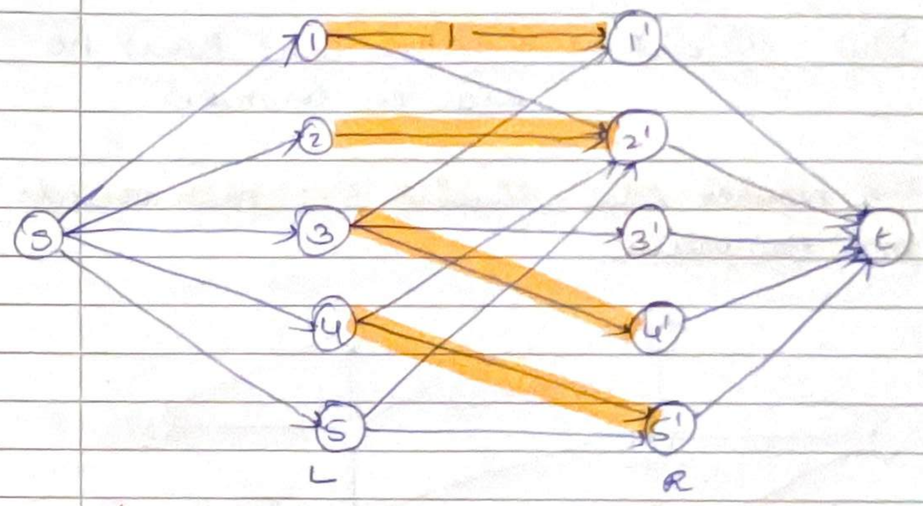
* Undirected Graph $G = (V, E)$
* $M \subseteq E$ is a matching if 1 node appears on only 1 edge.

Bipartite Matching

* $G = (L \cup R, E)$
* $M \subseteq E$
* Max Matching : find max cardinality



To find Max Matching = find Max flow.



Max Cardinality Matching = Max flow

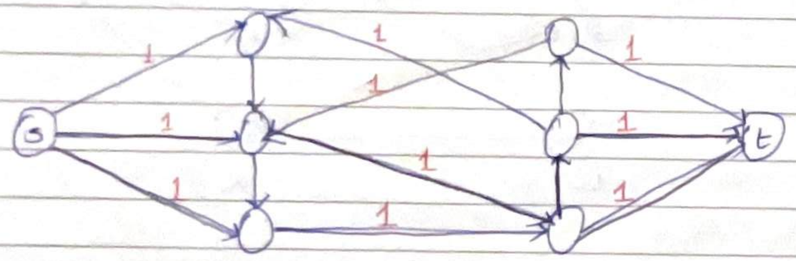
- Given matching M of cardinality k
  flow f sends 1 unit along k paths
  f has value k
  value of f ≤ value of max flow in G'
- Let f = max flow in G' of value k.
  k can have value f(e) 0 - 1
  M' = set of edges from L to R = 1
  each node participates in 1 edge in M'
  $|M'| = k$

* Running time = $O(mn)$ $c = n$ here

Disjoint paths

* Edge - disjoint : = 2 paths that have no edge en common.

* Max number edge - disjoint s - t path equals max flow value.



* Running time = $O(mn)$, if $c=n$.

Network connectivity

* A graph $G=(V,E)$, 2 nodes s & t.
find 2 edges whose removal will disconnect t from s.

$F \subseteq E$, disconnects s & t.

Menger

The max number of edge - disjoint $s-t$ paths = min number of edges whose removal disconnects t from s

Extension of max flow

* **Circulation**

A function that satisfies

- For each $e \in E : 0 \leq f(e) \leq c(e)$ → capacity cond
- For each $v \in V : \sum f(e) - \sum f(e) = d(v)$ → demand condition

* A Graph has circulation

iff G' (with s & t) has Max flow = D

D = Sum of supplies - sum of demand

* NO circulation

iff there exist a cut (A,B) such that.

$D > Cap(A,B)$.

P Decision problem for which there is a poly algo

NP Decision Prob for which there is a poly certifier

if $X \leq_p Y$ can be solved in poly time then
X can be solved in poly time.

Set Cover $\geq_p$ Vertex Cover $\equiv_p$ IS $\leq$ Set packing.

Set Cover $\equiv_p$ Set Packaging

3-SAT $\leq_p$ IS $\equiv_p$ Vertex Cover $\leq_p$ Set Cover.

X being reduced to Y
↳ $X \leq_p Y$

NP and Computational Intracibility

(Chapter -8)

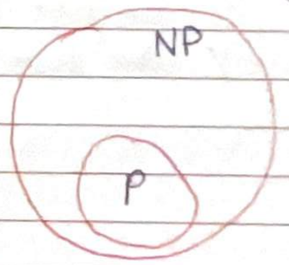
→ polynomial time → exponential time

linear search - n
Binary search - log n
Insertion sort - n^2
Merge sort - n log n
Matrix Multiplication

0/1 knapsack - 2^n
Traveling SP - 2^n
Hamiltonian - 2^n

P: Deterministic polynomial time taking.

NP: Non-Deterministic polynomial time



$\Rightarrow P \subseteq NP$

ExP: Exponential time algorithms

$NP \subseteq ExP$

NP-complete non deterministic, but has algo

- A prob 'Y' in NP with property every prob 'x' in NP $X \leq_p Y$

Circuit-SAT is NP complete.

$X \leq_p Y$.

* Make sure Y is atleast as hard as X.

Design algorithm

If $X \leq_p Y$, Y can be solved in poly-time
X can be solved in poly time.

Intractability

If $X \leq_p Y$, X can not be solved then
Y can not be solved.

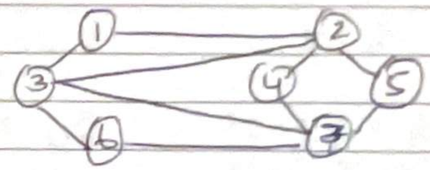
equivalence

$X \leq_p Y$ and $Y \leq_p X$ then $X \equiv_p Y$.

Independent set

$G = (V, E)$, find subset S of V that dont have edge b/w them

$IS = \{1, 4, 5, 6\}$



* Decision ⇒ Does there exists
* Optimization ⇒ find IS of max cardinality

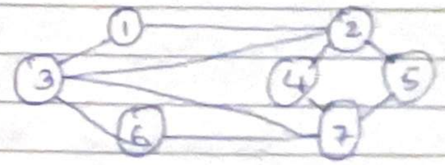
search p ≡p decision prob

Vertex Cover Decision Prob

* Set of vertices that cover all the edges.

$|S| \leq k$ & $S \subseteq V$

$VC = \{3, 2, 7\}$



Vertex cover $\equiv_p$ INDEPENDENT SET

S is an independent set iff V − S is a vertex cover

Special Case to general case

__/__/__

Set Cover

eg: $U = \{1, 2, 3, 4, 5, 6, 7\}$

$K = 2$

$S_1 = \{3, 7\}$ $S_4 = \{2, 4\}$

$S_2 = \{3, 4, 5, 6\}$ $S_5 = \{5\}$ $S_3 = \{1\}$

$S_6 = \{1, 2, 6, 7\}$

Vertex Cover $\leq_p$ set Cover

$S_2 \cup S_6 = U$

Set Packing

U = elements, k integer.

$S_1, S_2, \ldots S_m$

Setpacking $\geq k$, where these sets dont intersect.

__/__/__

Recipe to establish NP-completeness of Problem Y.

1) Show that Y is in NP.
2) Choose an NP-complete problem X.
3) PT $X \leq_p Y$ (not the reverse)

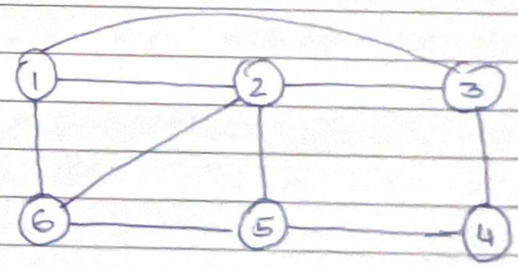
If you reduce X to Y

This means that Y is as hard as X

5
20
25

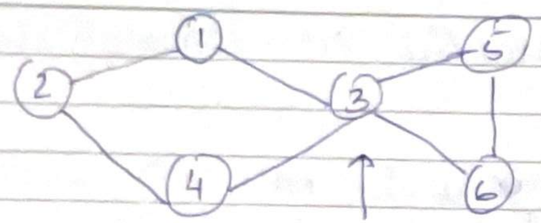
→ Hamiltonian Cycle (even number of nodes) for bipartite

* NP - Hard prob



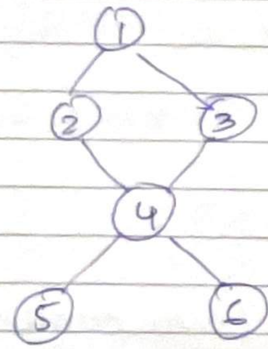
1, 2, 3, 4, 5, 6
1, 6, 5, 4, 3, 2, 1
1, 2, 6, 5, 4, 3, 1

* Changing starting point does not change cycle



Articulation point

These can not form MC



→ pendent nodes.

Decision Prob: Does there exist.

Optimization: find IS of max cardinality.

Optimization prob/ Search Prob $\equiv_p$ decision Prob

* if you solve optimization Prob, then find k^* and answer the decision Prob

* if you solve decision prob, then search over all values of k & find the maximum k for which answer is YES.

Set Covering

Try to cover ~~problems~~ a set of objects, using small sets

Set packaging

pack a large number of objects, subject to conflict

Jaivesh

Avinash

Dynamic Programming

DP problems are solved using sequence of decisions

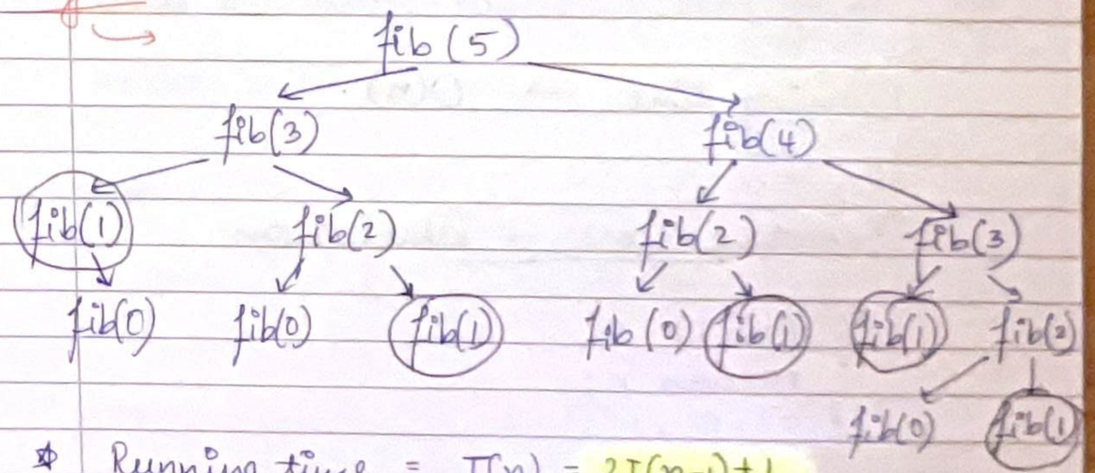
↓

The principal of optimality

fibbonci series

$$fib(n) = \begin{cases} 0 & \text{if } n=0 \\ 1 & \text{if } n=1 \\ fib(n-2)+fib(n-1) & \text{if } n>1 \end{cases}$$

Tracing tree



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

masters theorem = $O(2^n)$

* fib(1) called multiple times, so instead we store the value

Memoization → top-down approach.

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

* initaly -1.
* 5 → 3 → 1 (here we get fib(1) = 1) mark it in the array.
* fib(2) → 2 → 0 (gets 0) we have 1 already
  ↳ 0 + 1 = 1
* fib(3) → fib(1) + fib(2) = 2

⇒ no of calls changed from 15 to 6.

Running time → O(n).

Iterative func ⇒ Tabulation

```
if (n<1)
   return n;
F[0]=0 , F[1]=1

for(int i= 2 ; i<=n; i++)
   {
     F[i] = F[i-2] + F[i-1];
   }
   return F[n];
}
```

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

* We fill the table using iterative approach and start from smaller value
  ↳ Bottom-up approach

6.1) Weighted Interval scheduling

* Jobs → (start, finish, weight)
  - find the subset with maximum weights
  - The jobs are disjoint.
* Sort jobs according to finish times.

start with the last job

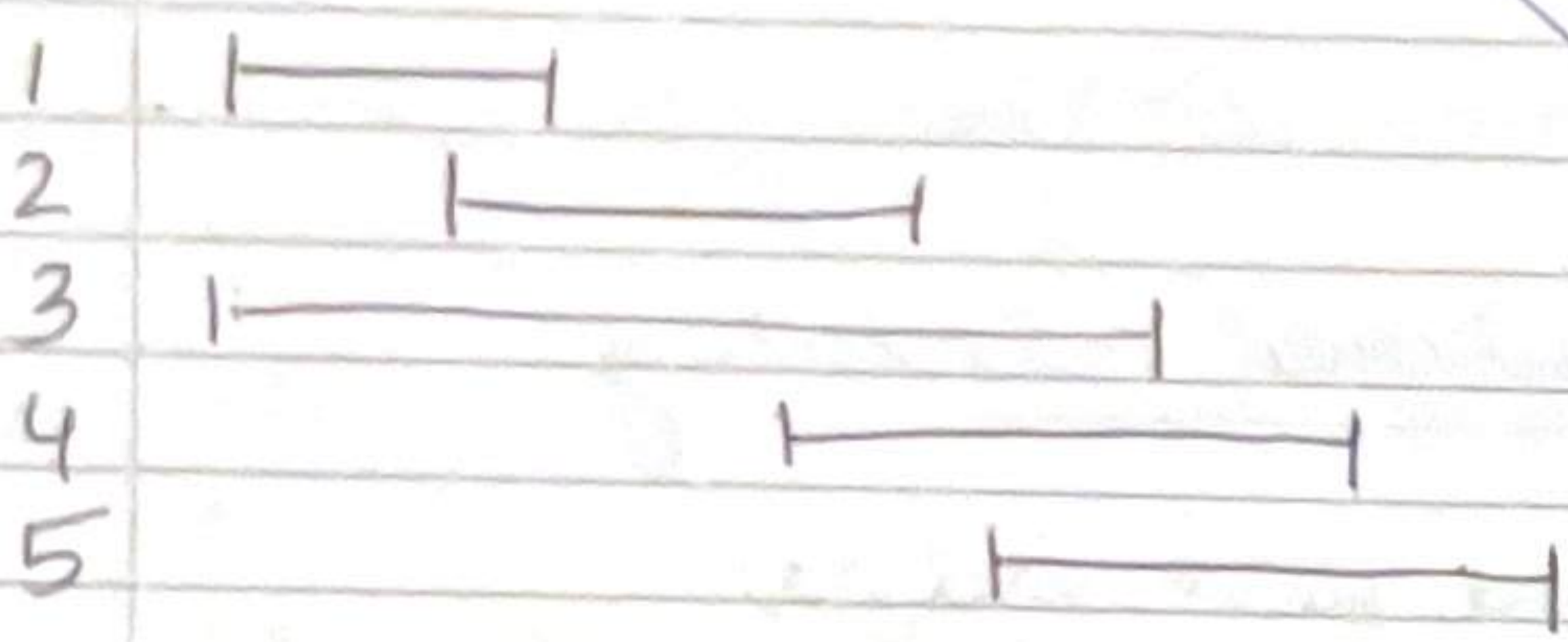
Case 1: (We pick it)
Then remove all overlapping jobs & compute solutions of jobs left

Case 2: (We dont pick it)
Then compute optimal solution for n-1 items

find OPT(J) = 'Max weight of non-overlap intervals'

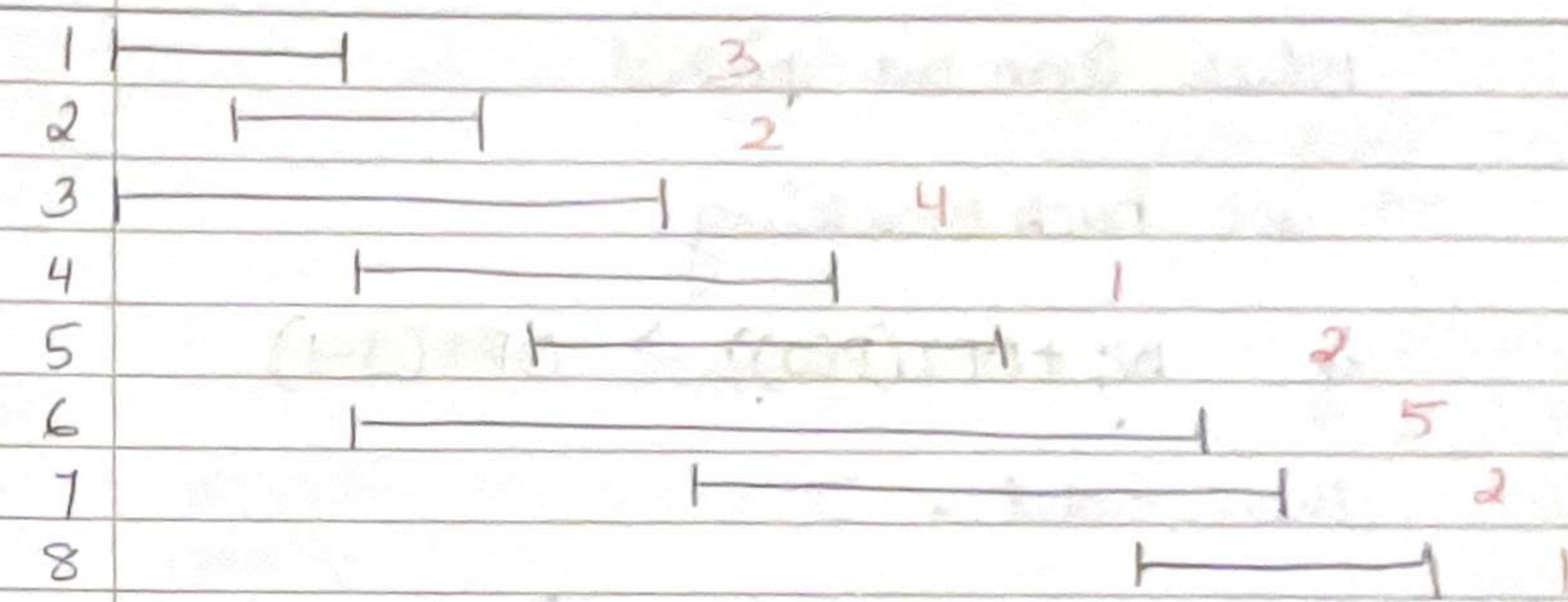
helps remove overlapping jobs

$P(J)$ = largest index i such that $f_i \le s_j$



$P(4) = 1$ (Job 1)
$P(5) = 2$ (Since Job 2 is larger index)

$$OPT(J) = Max\{w_j + OPT(P(J)),\ OPT(J-1)\}$$



J	0	1	2	3	4	5	6	7	8
w_j	0	3	2	4	1	2	5	2	1
P(J)	0	0	0	0	1	2	1	3	5
OPT(J)	0	3	3	4	4	5	8	8	8

1) $OPT(1) = max\{w_j + OPT(P(J)),\ OPT(J-1)\}$
$= max\{3+0, 0\} = 3$

3 is greater than 2+0

2) $OPT(2) = max\{2+0, 3\} = 3$

3) $OPT(3) = max\{4+0, 3\} = 4$

4) $OPT(4) = max\{1+3, 4\} = 4$

5) $OPT(5) = max\{2+3, 4\} = 5$ — 5 greater than 4

6) $OPT(6) = max\{5+3, 5\} = 8$ —

7) $OPT(7) = max\{2+4, 8\} = 8$

8) $OPT(8) = max\{1+5, 8\} = 8$

The value changes for the first time where max value = 8, ∴ Job 6 & P(6) are selected

Which item we picked

→ use backtracking

if $W_j + OPT(P(J)) > OPT(J-1)$

then add, J

else $OPT(J-1)$.

Running time

→ Sorting + max weight + backtracking

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

$= O(n\log n)$

* This was bottom up approach.

* In every DP problem, the subproblem is a DAG

nodes → subproblem

edges → dependencies

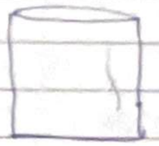
- weights - potential weights

6.2 → Knapsack Problem

(Subset sum)

$P = \{1, 2, 5, 6\}$

$W = \{2, 3, 4, 5\}$



$m = 8$ → weight constraint.

$n = 4$.

(a) $\max \sum x_i P_i$

(b) $\sum x_i w_i \leq m$.

→ Tabulation.

all weights →

P	W	Ob_j	0	1	2	3	4	5	6	7	8
			0	0	0	0	0	0	0	0	0
1	2	1	0	0	1	1	1	1	1	1	1
2	3	2	0	0	1	2	2	3	3	3	3
5	4	3	0	0	1	2	5	5	6	7	7
6	5	4	0	0	1	2	5	6	6	7	8

* fill last row using formula

$$V[i,W] = \max\{V[i-1,W], V[i-1, W-W[i]] + P[i]\}$$

$V[4,1] = \max\{V[3,1], V[3, 1-5] + 6\}$

$= \max\{V[3,1], V[3,-4]\} = V[3,1] = 0$ → not defined

⟹ This shows that till 5th weight I get same value as the row above

$V[4,5] = \max\{V[3,5], V[3,0] + 6\} = \max\{5, 6\}$

$= 6$

To find which objects for opt sol^n

→ sequence of decision

x_1 x_2 x_3 x_4

max { 0 1 0 1 }

W = 8. $8 \in 4$

$\notin 3$ obj 4 contributes

now 8 - profit of 4th obj

= 8 - 6

= 2

W = 2

$2 \in 3^{rd}$ row ∴ Obj 3 not needed

$2 \in 2^{nd}$ row.

$2 \notin 1^{st}$

∴ Obj 2 contributes.

now = 2 - 2

= 0

=

$0 \in 1$

$\in 0$ ∴ 1 does not contribute.

★ Running time $O(nW)$

pseudo polynomial - depends on numeric value of input instead of no. of input.

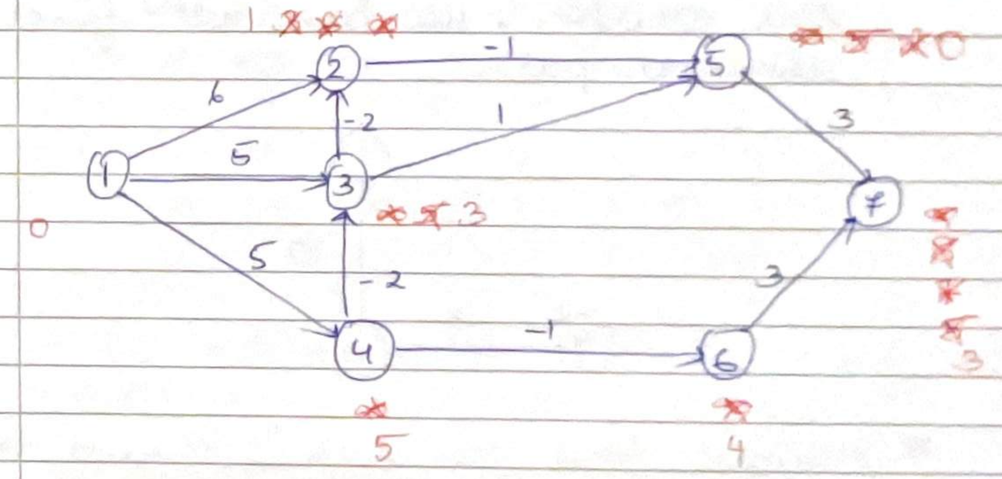
6.3) Shortest path

★ Bellman-ford

★ Relax n-1 times.

if $(d(u) + c(u,v) < d(v))$

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

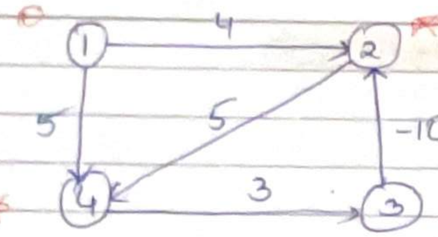


1 - 0
2 - 1
3 - 3
4 - 5
5 - 0
6 - 4
7 - 3

shortest path

★ Time complexity = $O(|V||E|) = O(n^2)$

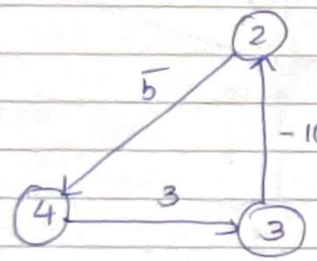
complete graph/dense = $O(n^3)$



1
2
3
4

(1,2)(1,4)(2,4)(3,2)(4,3).

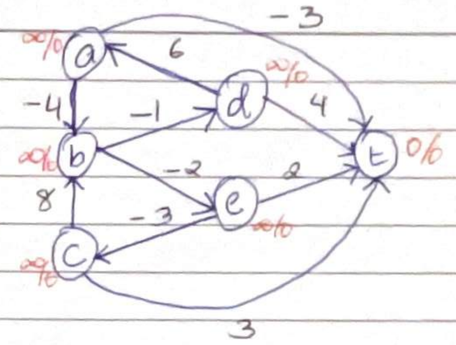
If there is a negative cycle in the graph, we can not find the shortest path.



* Bellman-ford can check if there is a neg cycle by performing 'n' iterations
* Running time – O(mn)

$$OPT(i,v) = \begin{cases} \infty & \text{if } i = 0 \\ \min\{OPT(i-1,v), \min\{OPT(i-1,w) + c_{vw}\}\} & \end{cases}$$

Ex.



	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3	-3	-4	-6	-6
b	∞	∞	0	-2	-2	-2
c	∞	3	3	3	3	3
d	∞	4	3	3	2	0
e	∞	2	0	0	0	0

* start with 1 edge to 't', then move to nodes at a distance of 2 edges

$$OPT[v] = \min\{OPT[x], OPT[w] + c_{vw}\}$$