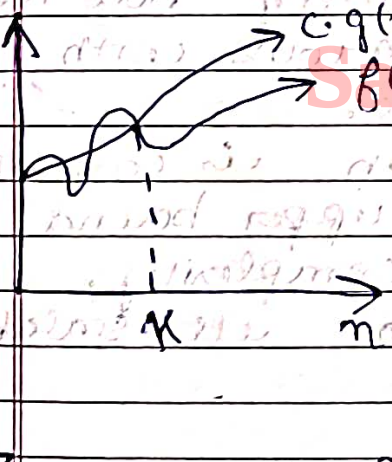
Design and Analysis of Algorithms :-

Characteristics of algorithm

(i) Input → 0 or more well defined inputs
(ii) Output → must generate atleast one output.
(iii) Unambiguous. | Definiteness → Read a,
Read b.
(iv) Finiteness :-
(v) effectiveness :- ex) Start
read a, b
Sum = (a+b)
Print Sum
end.

Asymptotic Notation

(1) Big - O.



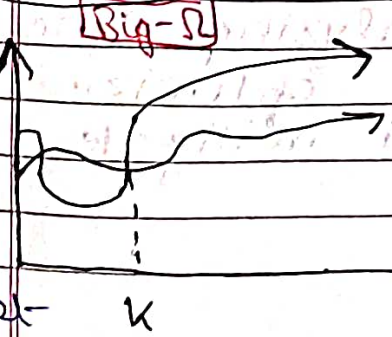
Worst case
(upper bound)
(At - most)

$f(n) = O(g(n))$
$f(n) \leq c \cdot g(n)$
$c > 0$
$n \geq k$
$k \geq 0$

ex) $f(n) = 2n^2 + n$; $f(n) = O(\ )$

$2n^2 + n \leq c \cdot g(n^2)$
$2n^2 + n \leq 3n^2$
$n \leq n^2$
$1 \leq n$
so $n \geq 1$ $k = 1$

(2) Big - Ω



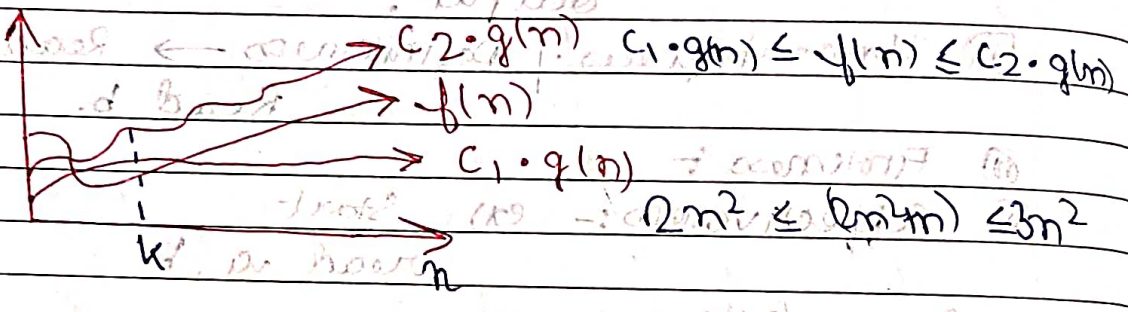
→ Best case
→ Lower Bound (At-least)

$f(n) = \Omega g(n)$
$f(n) \geq c \cdot g(n)$
$2n^2 + n \geq 2n^2$

Boundary

$n \geq 0$, $k = 0$

Theta (θ)



Performance measurement of Algorithms

1. **Time complexity analysis:** It assesses the computational time required by an algo as a fn of the size of its input. It focuses on understanding how the algorithm's runtime grows with increasing input size. $O(n)$ (Big O) notation is commonly used to express the upper bound of an algorithm's time complexity, providing insights into its scalability and efficiency.

2. Space complexity Analysis:- Space complexity analysis evaluates the amount of memory space consumed by an algorithm as a fn of the input size. It is essential in understanding memory requirements of an algo. especially in memory-constrained environments. By analysing space complexity, one can optimize memory usage and mitigate resource constraints.

③ Empirical analysis :- This involves empirically measuring the performance of algorithms through experimentation and observation. It entails running algo. on real-world or simulated data and measuring metrics such as execution time and memory usage.

④ **Tools and Techniques:** Various tools and techniques are available for performance measurement of algorithms, including profiling tools, debuggers, and performance analysis libraries. These tools aid in collecting performance data, identifying bottlenecks, and optimizing algorithm implementations for improved efficiency.

Time and Space tradeoffs :-

⑤ Tradeoff → It arises when one algo optimizes space complexity and other one optimizes time complexity, so a tradeoff (to accept a Ds advantage) arises between these two. It refers to the balancing act between optimizing the runtime efficiency (time) of an algo. and minimizing its memory consumption (space).

Recurrence Relation

10 20 30 40 50 60 70

i = 1 ↓ j = 7

$mid = \left(\frac{1+7}{2}\right) = 8/2 = 4$

Binary Search

```
BS(a,i,j,x)
    mid = (i+j)/2
    if(a[mid]==x)
        return(mid);
    else
        if(a[mid]>x)
            BS(a,i,(mid-1),x)
        else
            BS(a,(mid+1),j,x)
```

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

Substitution Method

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

$T(n) = T(n/2) + c$ — ①

$T(n/2) = T(n/4) + c$ — ⑪

$T(n/4) = T(n/8) + c$ — ⑪⑪

$\Rightarrow T(n) = T(n/4) + c + c$

$\Rightarrow T(n) = T(n/4) + 2c$

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

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

$= T\left(n/2^k\right) + kc$

$T(1) = 1$

now, $n = 2^k$

$\Rightarrow \log_2(n) = k \log_2(2)$

$\Rightarrow \boxed{k = \log_2(n)}$ → $T(n) = T(1) + kc$

$T(n) = \log_2(n) \cdot c$

$\boxed{O(\log_2(n))}$ (Ans)

② $T(n) = \begin{cases} 1 & \text{if } n=1 \\ n * T(n-1) & \text{if } n>1 \end{cases}$

$T(n) = n * T(n-1)$ — (i)

$T(n-1) = (n-1) \cdot T(n-2)$ — (ii)

$T(n-2) = (n-2) \cdot T(n-3)$ — (iii)

$\Rightarrow T(n) = n \cdot (n-1) \cdot T(n-2)$

$\Rightarrow T(n) = n(n-1) \cdot (n-2) \cdot T(n-3)$

$\vdots$

upto $(n-1)$ steps.

$\Rightarrow T(n) = n(n-1)(n-2)(n-3) \cdots [n-(n-1)]$

$\Rightarrow T(n) = n(n-1)(n-2)(n-3) \cdots * 1$

$T(n) = n \cdot n(1-1/n) \cdot n(1-2/n) \cdot n \cdot (1-3/n) \cdots n(3/n) \cdot n(2/n) \cdot n(1/n) \cdot$

$T(n) = (n \cdot n \cdot n \cdots n \text{ times}) \times (\ldots\ldots)$

Some terms where obviously Power of n^4 (and $4 < n$) so ignored

$T(n) = (n^n)$

$O(n^n)$

③ $T(n) = \begin{cases} 1 & \text{if } n=1 \\ 2T(n/2) + n & \text{otherwise} \end{cases}$ — (i)

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

$T(n/4) = 2T\left(n/2^3\right) + \left(n/2^2\right)$ — (iii)

$T(n) = 2\left[2T\left(n/4\right) + n/2\right] + n$

$= 2^2 \cdot \left[T\left(n/2^2\right) + n \right] + n$

$= 2^3 \left[T\left(n/2^3\right) + 3n \right]$

$\vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \quad (k \text{ times})$

$= 2^k \cdot T\left(n/2^k\right) + kn$

$\Rightarrow \frac{n}{2^k} = 1 \quad [T(1) = 1]$

$\Rightarrow \log_2 n = k(\log_2 2)$

$\Rightarrow \boxed{k = \log_2 n}$

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

$\therefore O(n \log n)$

Masters Theorem

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

$a \geq 1, \quad b > 1$

Soln is:

$$\boxed{T(n) = n^{\log_b a} \cdot [U(n)]}$$

$\Rightarrow U(n)$ depends on $f(n)$.

$$h(n) = \frac{f(n)}{n^{\log_b a}}$$

Relation between $h(n)$ and $U(n)$ is:

if $h(n)$	$U(n)$
$= n^a, \; a > 0$	$\Omega(n^a)$
$= n^a, \; a < 0$	$O(1)$
$(\log n)^i$	$\frac{(\log_2 n)^{i+1}}{(i+1)}$
$i \geq 0$	

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

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

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

$= n^{\log_2 8} \cdot U(n)$

$= n^3 \cdot U(n)$

$= n^3 \cdot O(1)$

$= n^3$

$O(n^3)$

$h(n) = \frac{n^2}{n^3} = (n-1)$

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

$T(n) = n^{\log_2(1)} \quad U(n)$

$= n^0 \cdot U(n)$

$h(n) = \frac{f(n)}{n^{\log_b^a}} = c$

so, $(\log_2 n)^0 \cdot c = h(n)$

then $U(n) = (\log_2 n)$

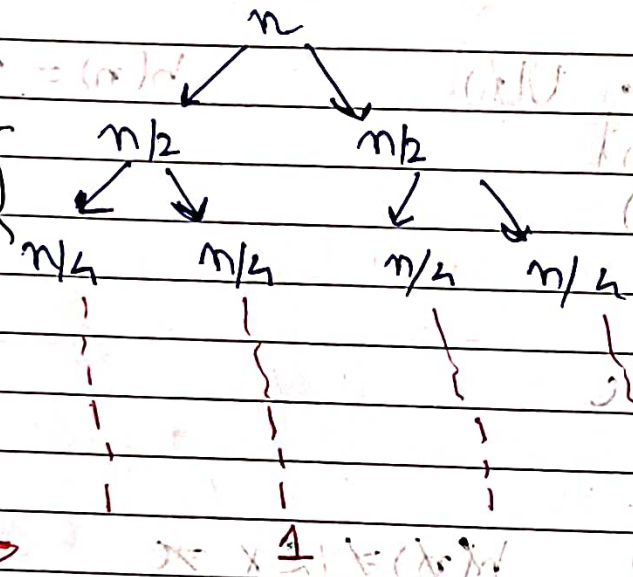
$= (\log_2 n) \cdot c$

$T(n) = (\log_2 n)$

$O(\log_2 n)$

Recursive Tree Method

① $T(n) = 2T(n/2) + cn$ → cost required at each step.

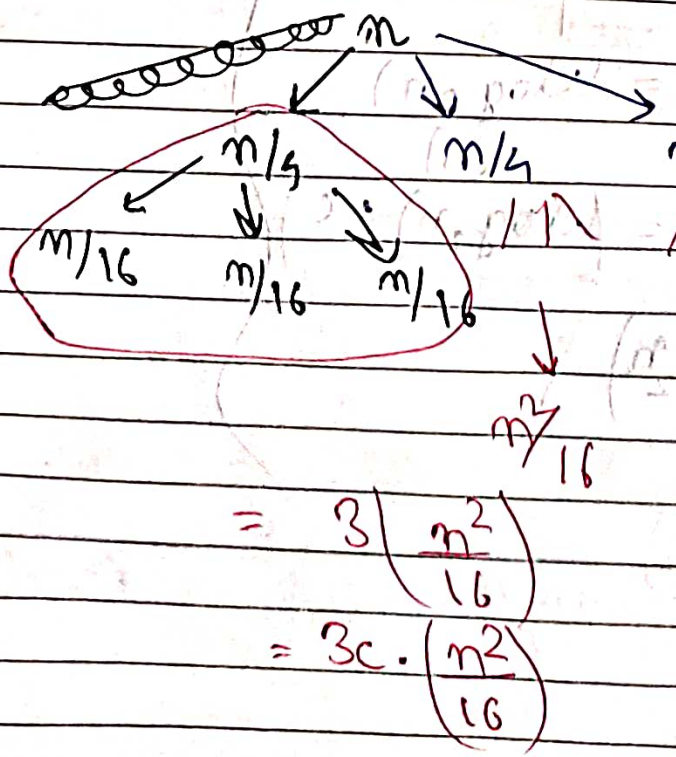


Height of the tree = $(Cn) \cdot (\log n)$

Since series is logarithmic.

$O(n \log n)$

② $T(n) = 3T(n/4) + cn^2$



$= 3\left(\frac{n^2}{16}\right)$

$= 3c \cdot \left(\frac{n^2}{16}\right)$

$$T(n) = cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 \cdot cn^2 + \left(\frac{3}{16}\right)^3 \cdot cn^2 + \dots$$

$$= cn^2 \left[1 + \frac{3}{16} + \frac{3}{16} + \dots\right]$$

$$= cn^2 \times \left(\frac{1}{1 - \frac{3}{16}}\right)$$

$$= cn^2 \cdot \frac{16}{13}$$

Revise this ②

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

Brute Force Algorithmic Strategy

→ It is a straightforward approach to solve problems by systematically examining all possible solutions and selecting the best one. It involves exhaustively searching through the entire soln space without any optimizati on or heuristics.

① Exhaustive Search

→ Brute force algos typically involve exhaustive search, where all possible solns are explored systematically. This often involves generating all possible combinations or permutations and evaluating each one to determine the optimal soln.

② Nested Loops:

In many cases, these algos use nested loops to iterate through all possible combinations of elements or choices. Each iteration represents a potential soln, which is then evaluated to determine its quality or correctness.

Ex) Subset Sum Problem, Travelling Salesman Problem, Knapsack Problem, N-Queens Problem, Subset Generation etc.

Greedy Method → used to solve optimisation problems.

```
Algorithm Greedy (a, n)
{
    for i=1 to n do
    {
        x = select(a);
        if feasible(x) then
            soln = soln + x;
    }
}
```

n=5

a	a_1	a_2	a_3	a_4	a_5
	1	2	3	4	5

→ this method states that problem is solved in stages, in each stage we consider an input from a given problem, if that is feasible, we will include it in the soln. By including all these feasible inputs, we will get a optimal soln.

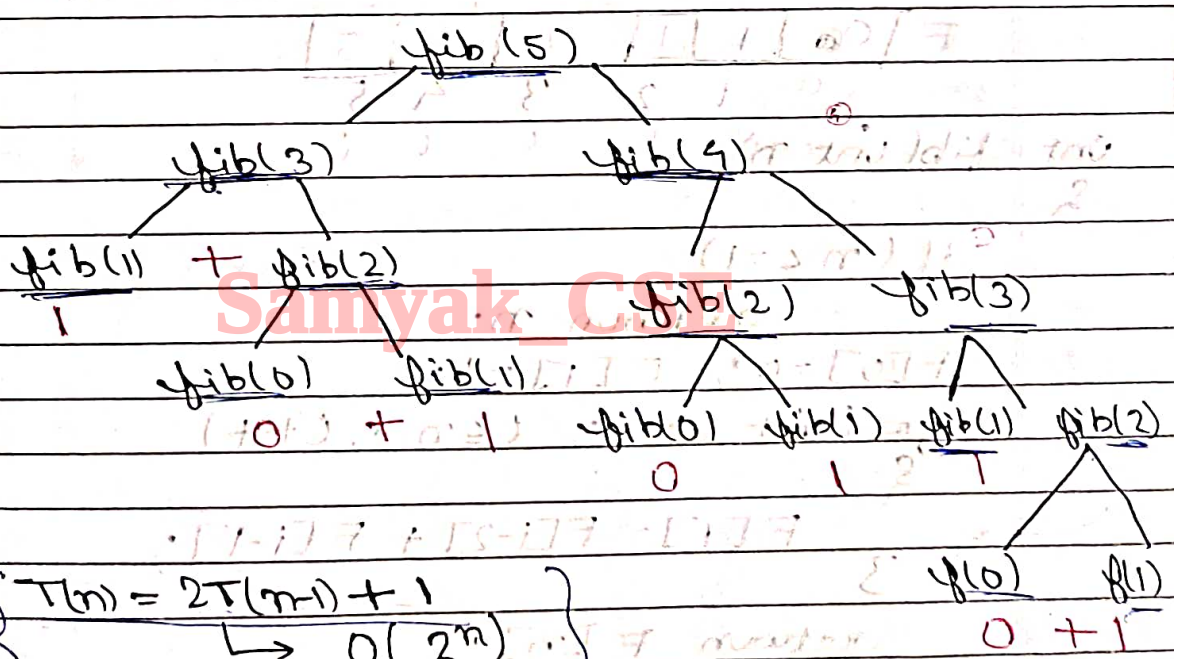
Dynamic Programming

↳ Searches for every possible soln. then picks the best soln.
→ uses more time.
→ follows principal of optimality.

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

```
int efib (int n)
{
  if (n<=1)
      return n;
  return fib(n-2) + fib(n-1);
}
```

0, 1, 1, 2, 3, 5, ...



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

⇒ To reduce this time complexity, we have another approach,

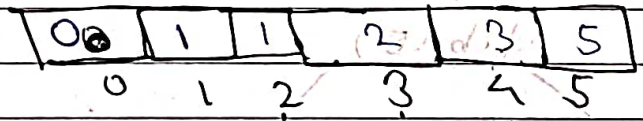
F	-1	-1	-1	-1	-1	-1
	0	1	2	3	4	5

When fib(5) is called, it is checked fib(5) available no, then go to fib(3) not available then fib(1) available set it to 1, then go to fib(2) then fib(0)=0 set it then fib(1)=1, already available, then add fib(1)+fib(2) give fib(3)=2, and so-on as soon as the value is available,

set those values in the array.
now, if 'fib(6) is called, then fib(4) and fib(5) is already available.

For fib(n) it's making (n+1) calls, so (n+1) means. O(n).
→ here we can see that memorization follows top-down approach.

Same Problem using Tabulation in Dynamic Programming.




```
int fib(int n);
{
    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];
}
```


Branch and Bound (For optimisation problems)

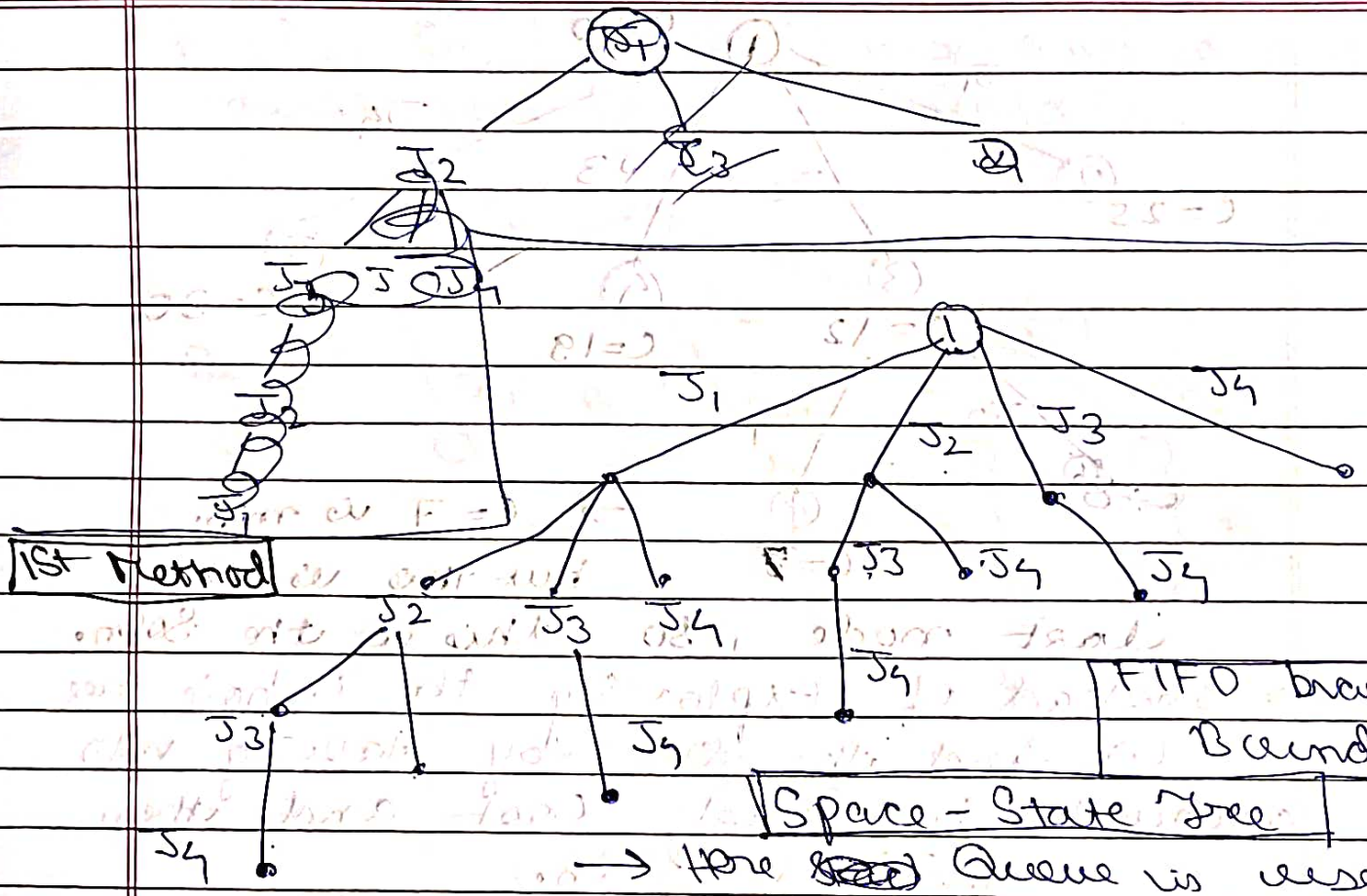
For minimization problems only.

Job Sequencing

Jobs = {J1, J2, J3, J4}
P = {10, 5, 8, 3}
d = {1, 2, 1, 2}

S1 = {J1, J4}
S2 = {1, 0P, 1}

Uses Brute Force approach.



1st Method

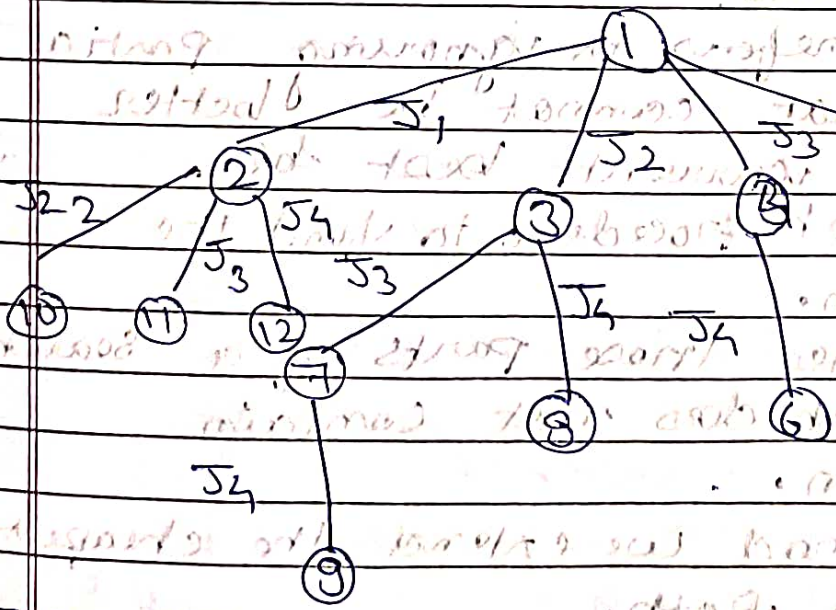
FIFO branch-Bound

Space-State Tree

→ Here Queue is used to implement this method.

2nd Method.

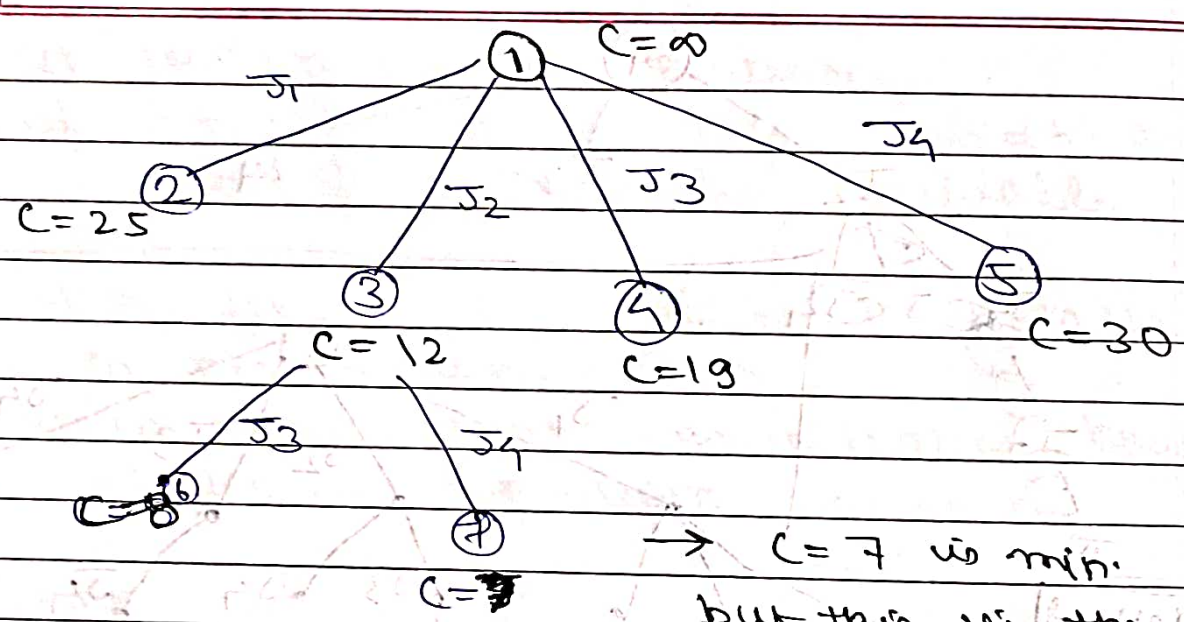
Here Stack is used to implement this method.



LIFO-Branch-Bound

3rd Method. LC-BB → Least Cost Branch and Bound

→ For every node, we will calculate the cost.



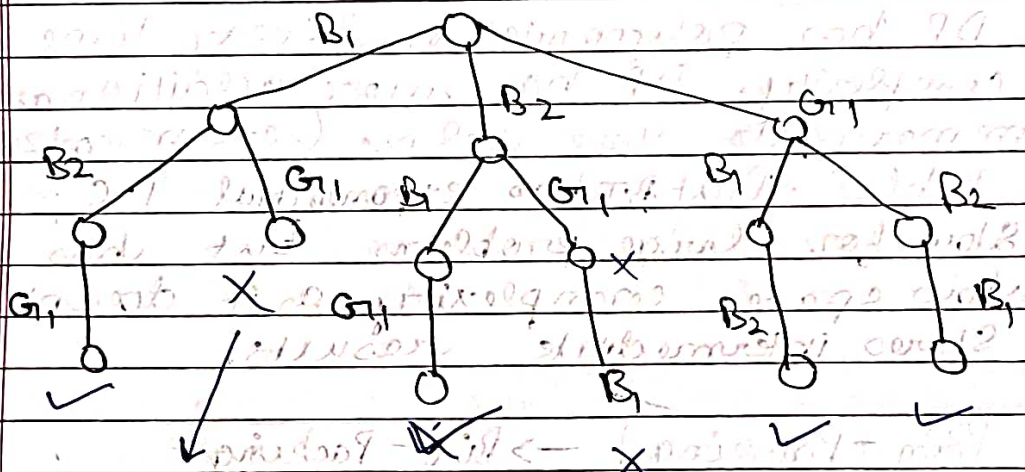
→ C=7 is min. but this is the last node, so this is the Soln.

→ Instead of exploring the whole tree we find the tree by taking into account the least cost and then find the Soln.

- Branching is the process of generating sub-problem.
- Bounding refers to ignoring partial solns. that cannot be better than the current best soln.
- It is a search procedure to find the optimal soln.
- It eleminates those parts of a search space which does not contain better soln.
- in this method we extend the cheapest partial path.

Backtracking (Brute Force Approach)
- → not used for optimisation.
- → It is used when we have multiple solns. and we want all those solns.

B_1, B_2, G_1 in 3 chairs and a girl must not be in the middle.



Killed bounding fn.

State Space Tree

→ Backtracking uses DFS and Branch and Bound uses BFS.

Working

1. Start with an initial soln.
2. Explore all possible extensions of the current soln.
3. If an extension leads to a soln, return that soln.
4. If not, backtrack to the previous soln. and try a different extension.
5. Repeat until all possible solns are explored.

→ DP allows overlapping subproblems (reusing solns.), while backtracking explores all possibilities without necessarily reusing solns.

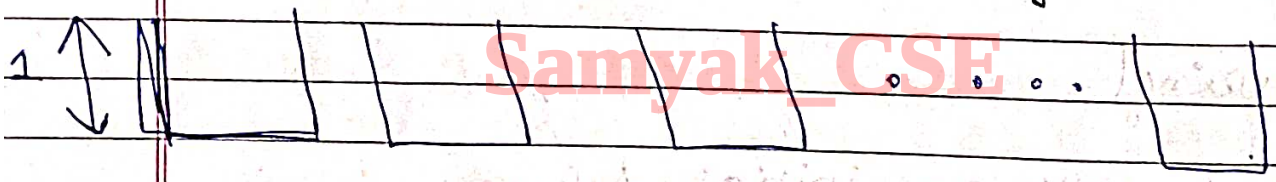
→ DP problems have optimal substructure (solving subproblems contributes to the

overall soln.) whereas backtracking doesn't always have this property.

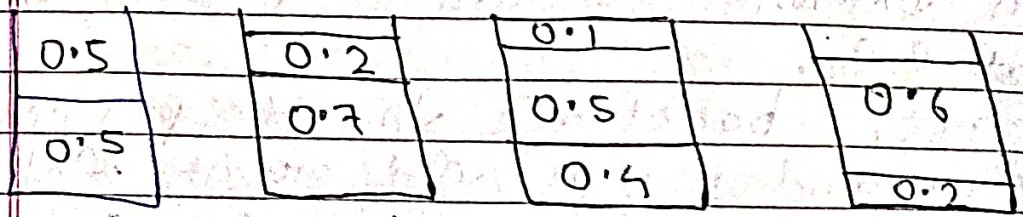
→ DP has polynomial or linear time complexity. DP has more additional memory to store solns. (ex memorization table). But BT has exponential T.C, slow for large problems, but has low space complexity, as it doesn't stores intermediate results.

Bin-Packing → Bin-Packing

→ Given 'n' items with sizes $s_1, s_2 \ldots, s_n$ such that $0 \le s_i \le 1$ for $1 \le i \le n$, pack them into the fewest no. of unit capacity bins.



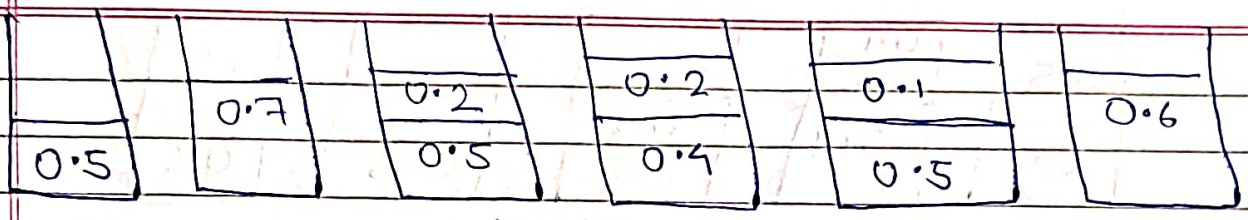
0.5, 0.7, 0.5, 0.2, 0.4, 0.2
0.5, 0.1, 0.6.



Mopt = 4 → but T.C is not in polynomial

Next Fit Algorithm ↓

Check to see if the current item fits in the current bin. If so, then place it there, otherwise start a new bin.

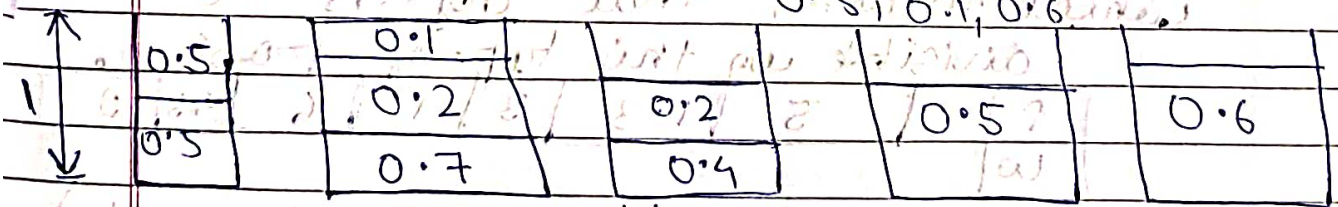


$M = 6$

Theorem Let M be the optimal no. of bins required to pack a list of I items. Next fit will use at most $2M$ bins. There exist sequences such that next fit uses $2M - 2$ bins.

First Fit Algorithm Scans the bins in order and place the new item in the first bin that is large enough to hold it. A new bin is created only when an item does not fit in the previous bins.

0.5, 0.7, 0.5, 0.2, 0.4, 0.2, 0.5, 0.1, 0.6

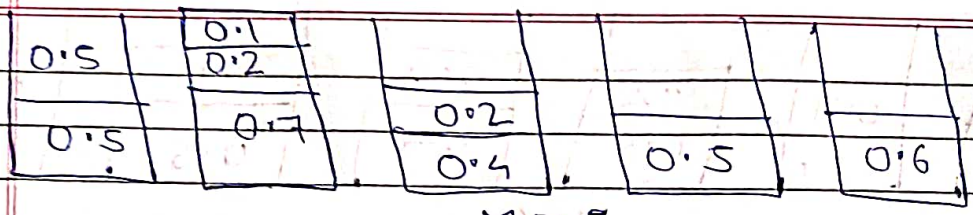


$M = 5$

Theorem → Let M = optimal no. of bins reqd. to pack I items. First fit will never use more than $1.7M$ bins. There exist sequences such that first fit uses $1.67M$ bins.

Best Fit Algorithm → New items placed in a bin where it fits the tightest. If it doesn't fit in any bin, start with a new bin.

0.5, 0.7, 0.5, 0.2, 0.4, 0.2, 0.5, 0.1, 0.6



n = 5

→ Can be implemented in O(n log n) time, by using a balanced binary tree storing bins ordered by remaining capacity.

Greedy Method → Knap Sack → optimization + maximization problem

n = 7, m = 15

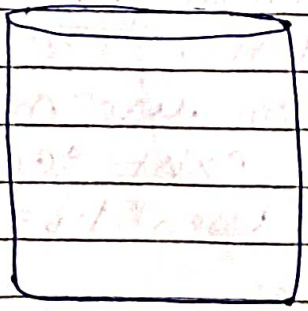
Objects O	1	2	3	4	5	6	7
Profits P	10	5	15	7	6	18	3
Weights w	2	3	5	7	1	4	1

→ here fractional values are also allowed, which means that objects are divisible in this type of problem.

P	5	1·3	3	1	6	4·5	3
w							

x	1	2/3	1	0	1	1	1
	x_1	x_2	x_3	x_4	x_5	x_6	x_7

$0 \le x \le 1$



15-1 = 14
14-2 = 12
12-4 = 8
8-5 = 3
3-1 = 2
2-2 = 0 ✗

eight

$= 1\times2 + \frac{2}{3}\times3 + 1\times5 + 0\times7 + 1\times1 + 1\times4 + 1\times1$

$= 2 + 2 + 5 + 0 + 1 + 4 + 1$

$= \boxed{15} \rightarrow \sum x_i w_i$

rofit $= 1\times10 + \frac{2}{3}\times5 + 1\times15 + 0\times7 + 1\times6 + 1\times18 + 1\times3$

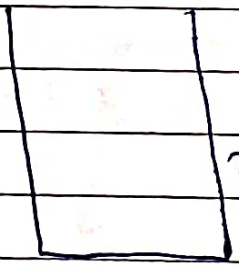
= 54.6

Constraint: $\sum x_i w_i \leq m$

Objective = $\sum x_i p_i$ = maximum.

Knapsack - Q-2.

Objects	obj	obj	obj
Profit	25	24	15
Weight	18	15	10



knap-sack capacity (m) = 20.

Algorithm

for i=1 to n
  calculate Profit / weight.
sort objects in decreasing order of P/W ratio.
for i=1 to n.
  if (m>0 and wi ≤ m)
    m = m-wi
    P = P+ Pi
  else break.
if (m>0)
  P = P+ Pi(m/wi);

→ O(nlogn) T.C

P/W =	1.3	1.6	1.5

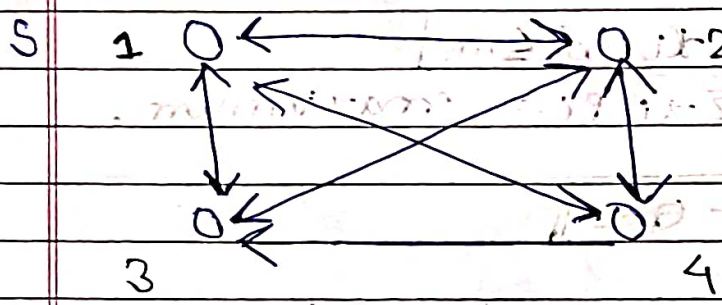
$(1 \times 24) + \frac{5}{10} \times 15$

= 24+7.5

= 31.5 (Ans)

Travelling Salesman Problem

: minimization problem



	1	2	3	4
1	0	10	15	20
2	5	0	25	10
3	15	30	0	5
4	15	10	20	0

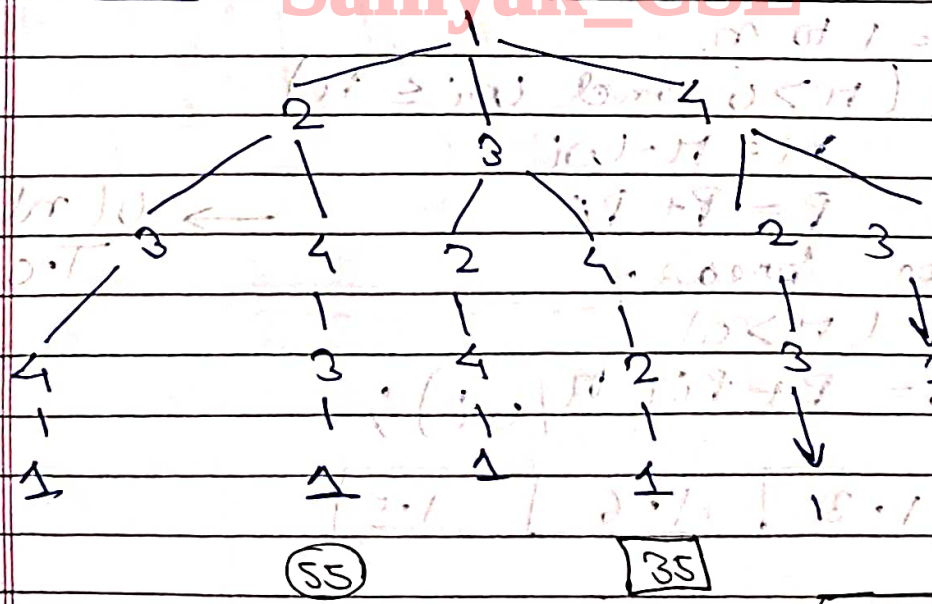
→ Distance / weight - edge matrix

Greedy Approach

1 → 2 (10) → 4 (+10) → 3 (+20) → 1 (+15)

= 55

Brute-Force



55 35

↳ $(n-1)!$ possibilities

$= (4-1)! = 3! = 6$

T.C = $O(n!)$

$\simeq O(n^n)$

Huge - T.C

$g(1, \{2,3,4\})$

$= \{ \min \{ c_{1k} + g(k, \{2,3,4\} - \{k\}) \} \}$

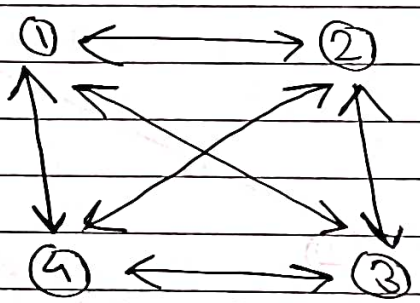
UQ 2,3,4

Dynamic Programming

→ TSP is a NP-complete Problem.

TSP → with Dynamic Programming

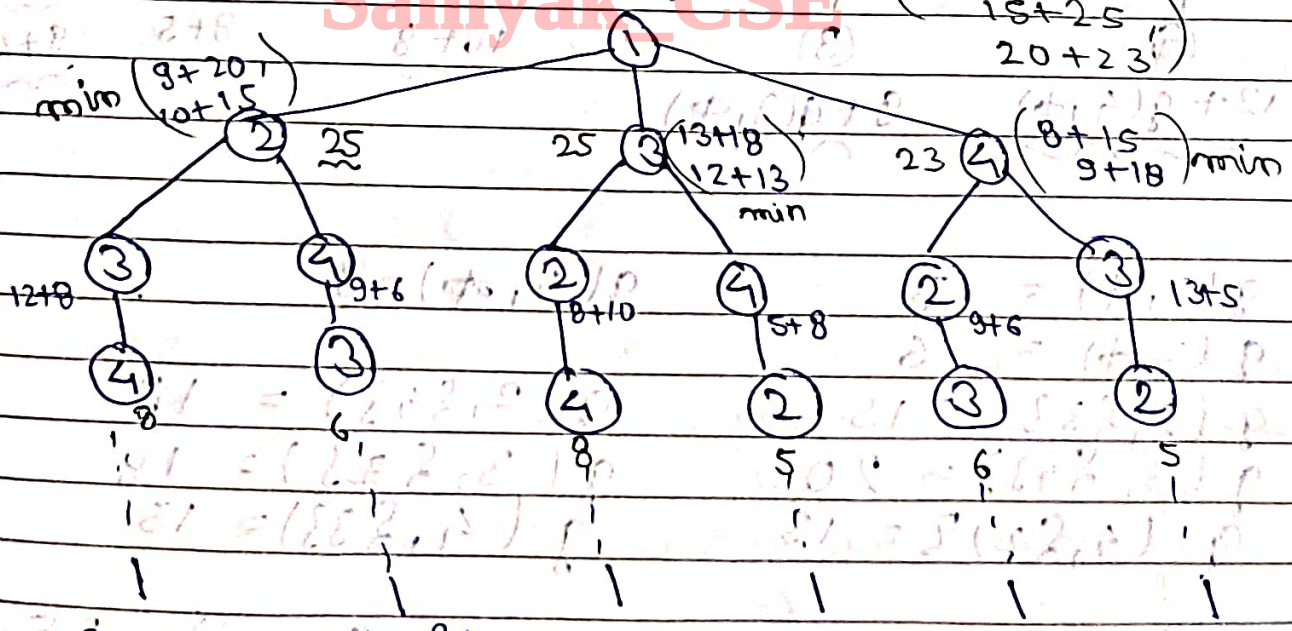
$$A = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ 1 & 0 & 10 & 15 & 20 \\ 2 & 5 & 0 & 9 & 10 \\ 3 & 6 & 13 & 0 & 12 \\ 4 & 8 & 8 & 9 & 0 \end{array}$$



General Formula with DP:

$$g(i,S) = \min_{k \in S} \{ c_{ik} + g(k, S-\{k\}) \}$$

- $g(i,S)$ → cost
- Starting Vertex
- Set of vertices



So, cost of shortest route is 35.

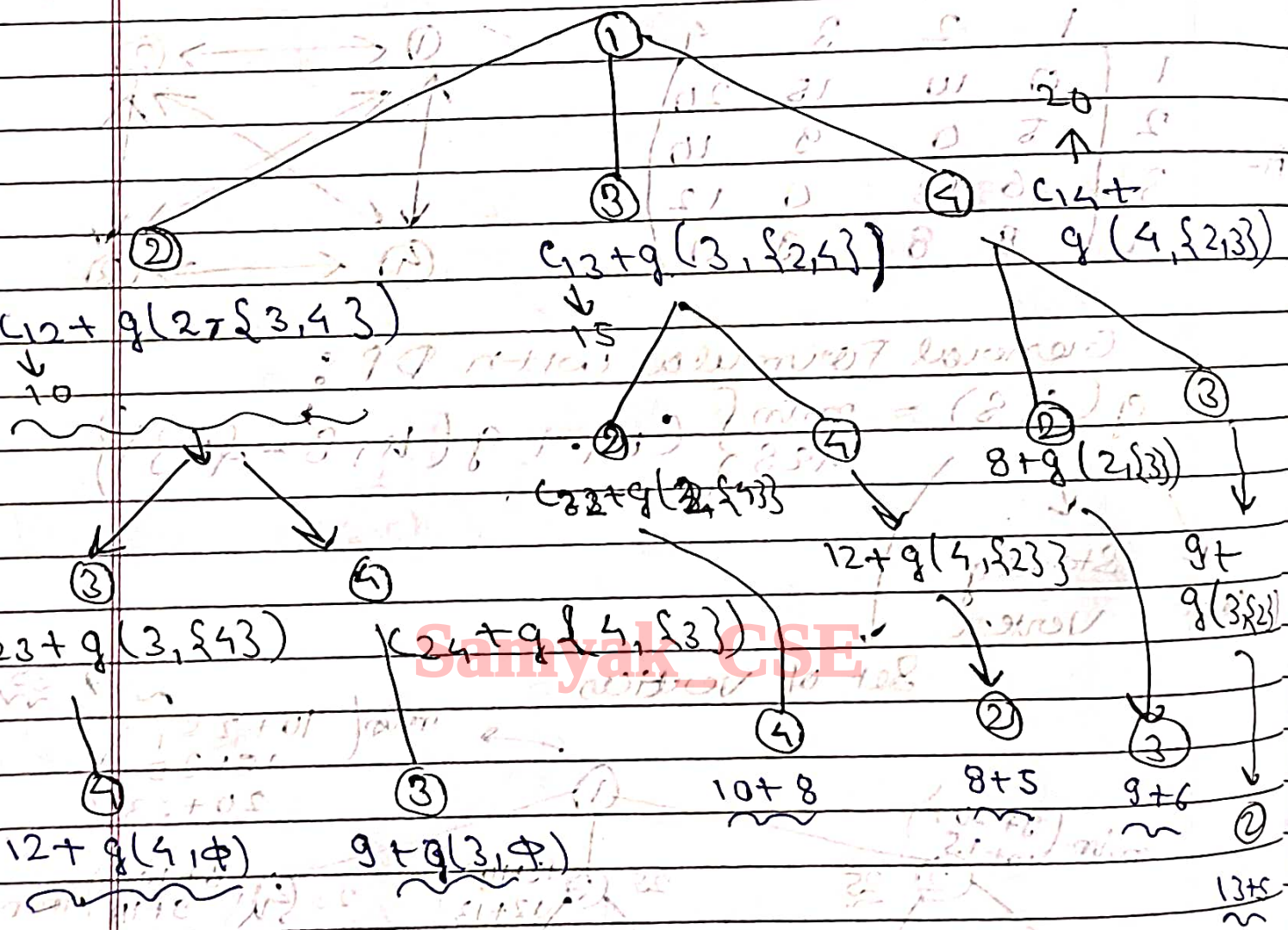
C_{ik} = cost of any vertex from 1 to k.

a) $g(2,\phi) = c_{21} = 5$

$g(2,\phi) = 6$

TSP - Formula Approach

→ $g(1,\{2,3,4\}) = \min_{k \in \{2,3,4\}} \{ c_{1k} + g(k, \{2,3,4\} - \{k\}) \}$



$g(2,\phi) = 5$
$g(3,\phi) = 6$
$g(2,\{3\}) = 15$
$g(3,\{4\}) = 20$
$g(4,\{2\}) = 13$

$g(4,\phi) = 8$
$g(2,\{4\}) = 18$
$g(3,\{2\}) = 18$
$g(4,\{3\}) = 15$

$g(2,\{3,4\}) = \underline{25}$, $g(3,\{2,4\}) = \underline{25}$
$g(4,\{2,3\}) = \underline{23}$

$$\boxed{g(1,\{2,3,4\}) = 35} \quad \text{(Ans)}$$

"Heuristics characteristics and Application domains in DAA."

→ Heuristics, in the context of algorithm design, refer to techniques or methods that prioritize finding satisfactory solns. quickly, even if they are not guranteed to be optimal.

① Feasibility → They are often designed to find feasible solutions efficiently. They prioritize practicality over optimality, making them suitable for real-world problems where finding an optimal soln might be computationally expensive.

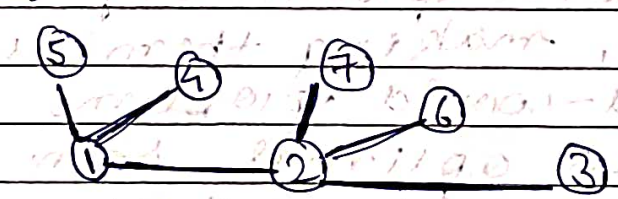
② Approximation → These algos provide approximate solns to complex problems. While these solns may not be optimal, they are generally acceptable within a certain margin of error.

③ Speed → These algos are designed to be fast and efficient. They aim to quickly converge towards a satisfactory soln without exhaustive search. This makes them suitable for large-scale optimization problems where computational resources are limited.

④ Simplicity → They are usually simpler and easier to implement compared to exact algos.

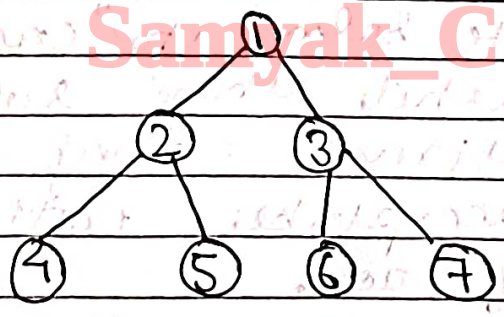
⑤ **Adaptability** → They can adapt their strategies based on feedback or changes in the problem environment. This adaptability is crucial in dynamic problem-solving scenarios in DAA, such as in online algos for network routing or scheduling, where the system's state evolves over time.

Depth First Search DFS



(i) visiting a vertex
(ii) exploration of a vertex

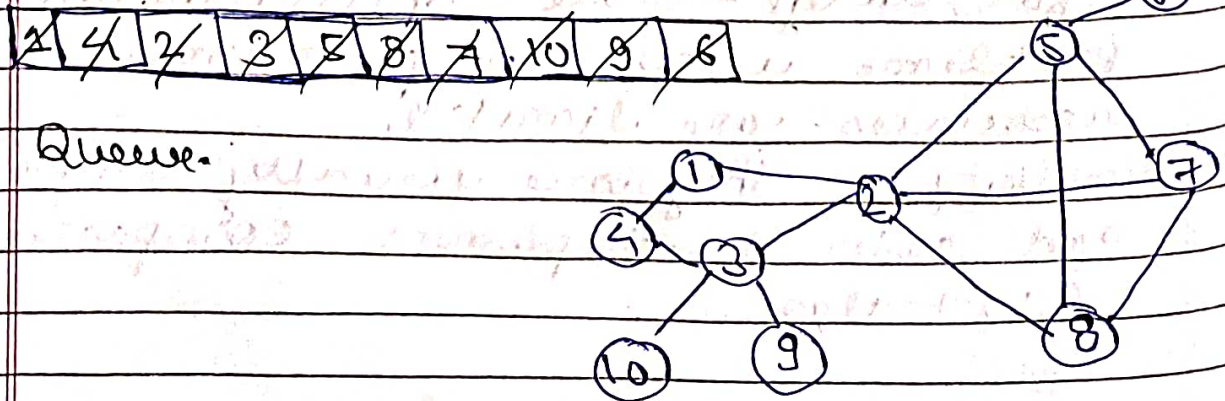
(any order) BFS: 1, 2, 4, 5, 7, 3, 6
DFS: 1, 2, 3, 6, 7, 4, 5

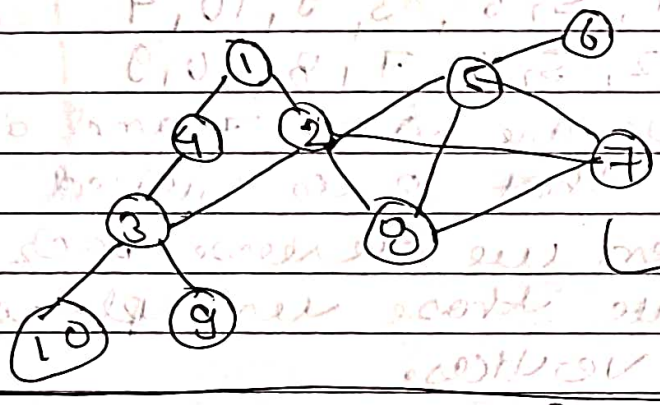


BFS: 1, 2, 3, 4, 5, 6, 7
BFS is just like level order in a binary tree.
DFS: 1, 2, 4, 5, 3, 6, 7. (Pre-order)

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

Q → Queue.





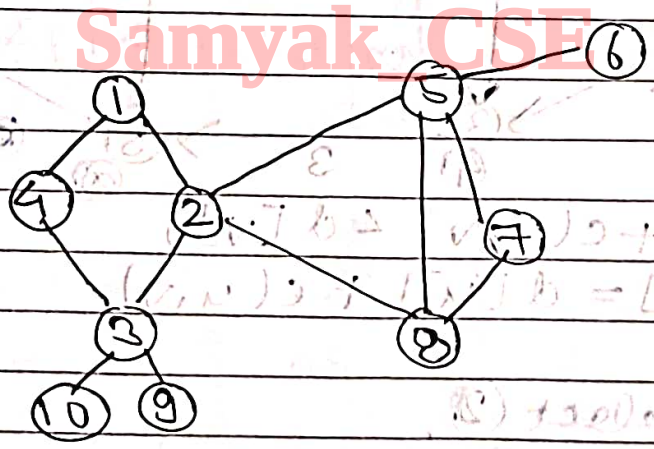
BFS
↳ Spanning Tree.

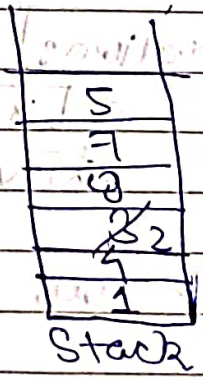
→ When 9 visit ①, 9 must select ④ and ②, after that only 9 can select ④ for exploration..
valid

→ Other BFS are:
↳ 1, 2, 4, 5, 7, 8, 3, 10, 9, 5, 6.

↳ 5, 2, 8, 7, 6, 3, 1, 9, 10, 4

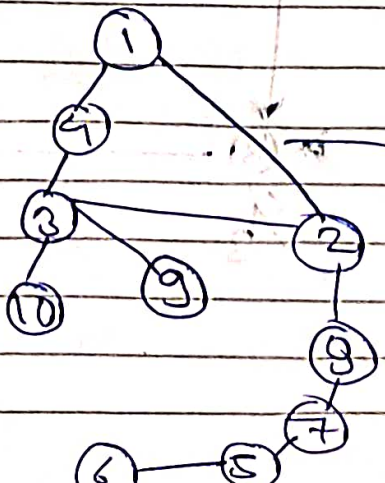
Depth First Search DFS





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

DFS Spanning Tree



→ back edges.

(Pre-order traversal)

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

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

Suppose we are at 7 and after that 8 is visited then we reverse back and complete those unexplored vertices.

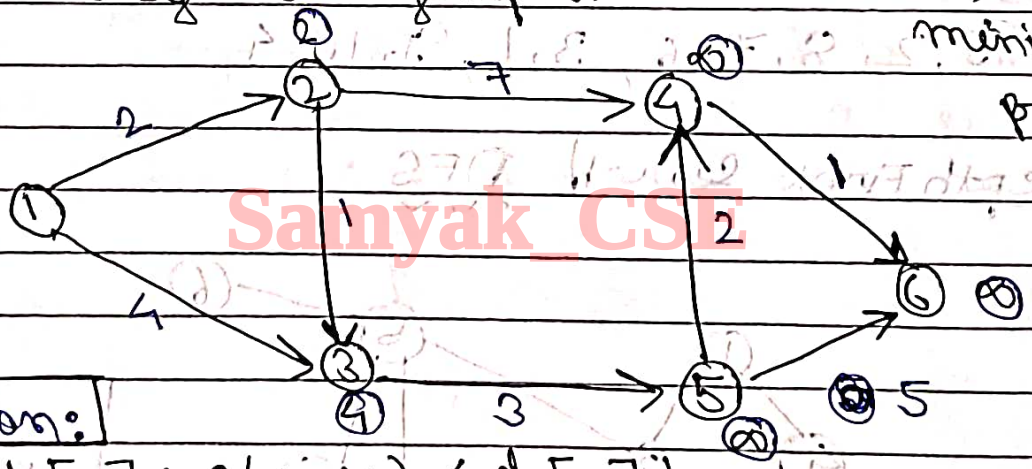
T.C of BFS, DFS = O(n)

n = no. of vertices.

Single Source Shortest Path (Dijkstra's Algorithm)

→ For weighted graphs.

minimization problem.

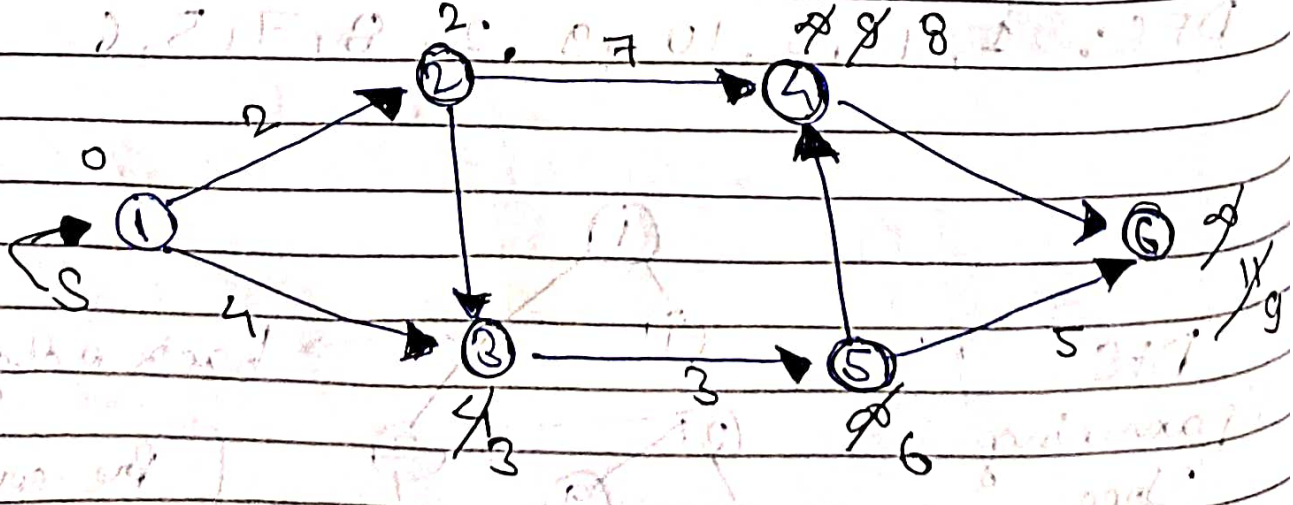


Relaxation:

if [d[u] + c(u,v) < d[v]]

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

→ now select (2).



V	d[v]
2	2
3	3
4	8
5	6
6	9

$n = |V|$ $\quad |V|$

$n \times n$ → atmost 'n' vertices can be relaxed (depends on the nature of the graph)

For 'n' Vertices

$O(n^2)$

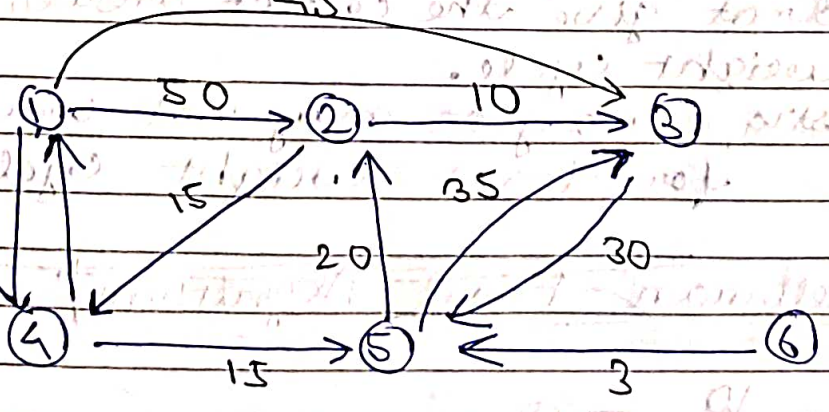
↓

Worst Case.

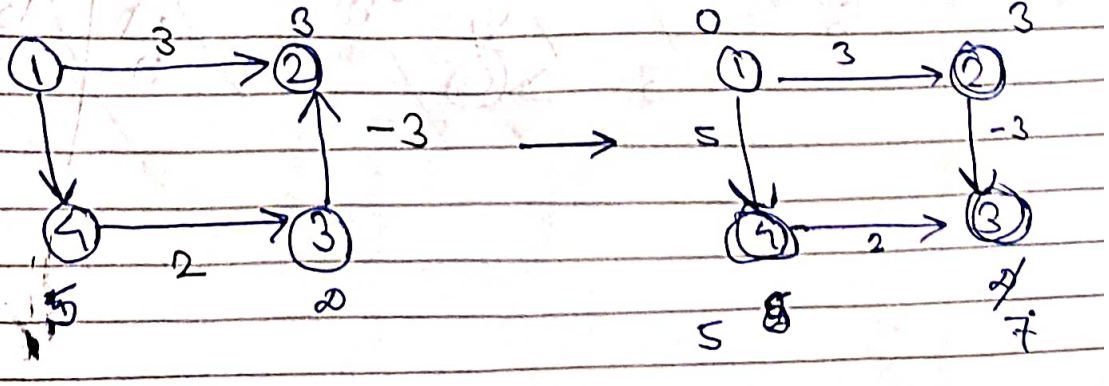
$O(|V|^2)$ → Worst Case T.C

Starting Vertex 1

Selected Vertex	2	3	4	5	6
4	50	45	(10)	∞	∞
5	50	45	10	(25)	∞
2	45	45	(10)	(25)	∞
3	(45)	(45)	(10)	(25)	(∞)
6	45	45	10	25	∞

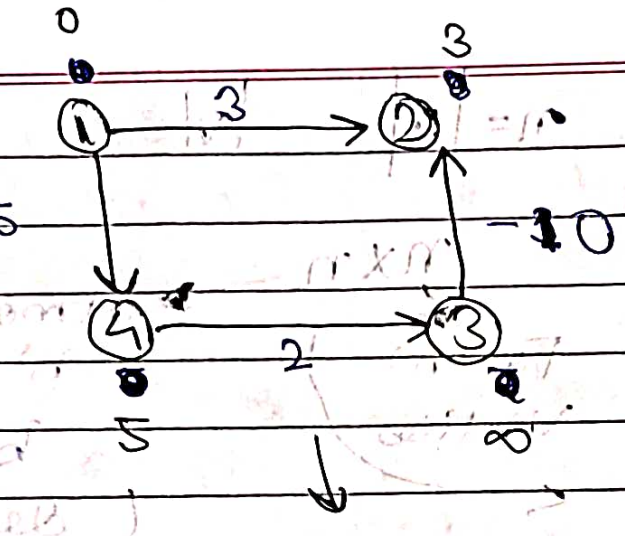


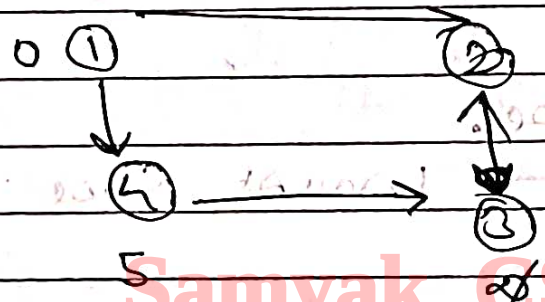
Drawback of Dijkstra's algorithm



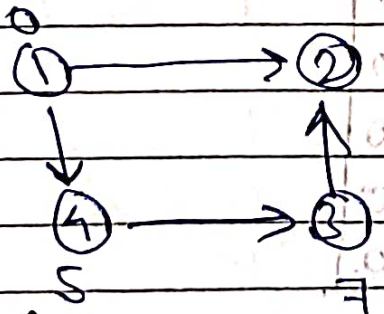
Works

now,





now dijkstra Stops, here but due to (-ve) weight cycle, if we again check, then

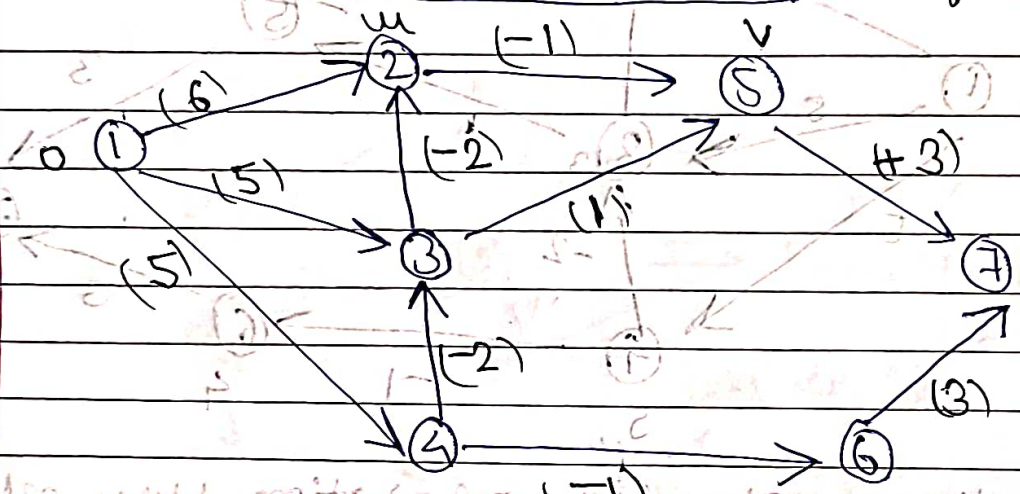


→ More Shorter Path, so dijkstra has failed, it couldnot give the correct answer due to (-ve) weight cycle.

→ Dijkstra, may or may not work for (-ve) weight cycle.

Single-Source Shortest Path

Bellman-Ford :→ follows DP.



→ Relax the edges (n-1) times, n = no. of vertices.
↳ 6 times.

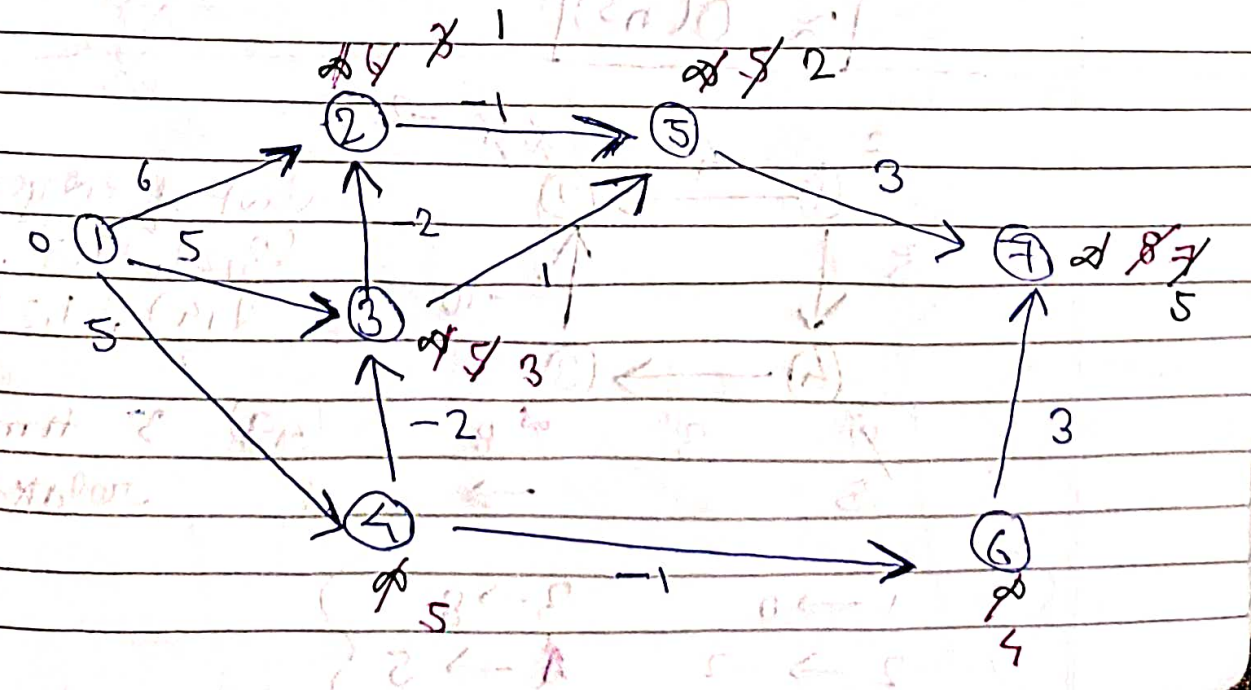
|V| = n = 7.

Relaxation:

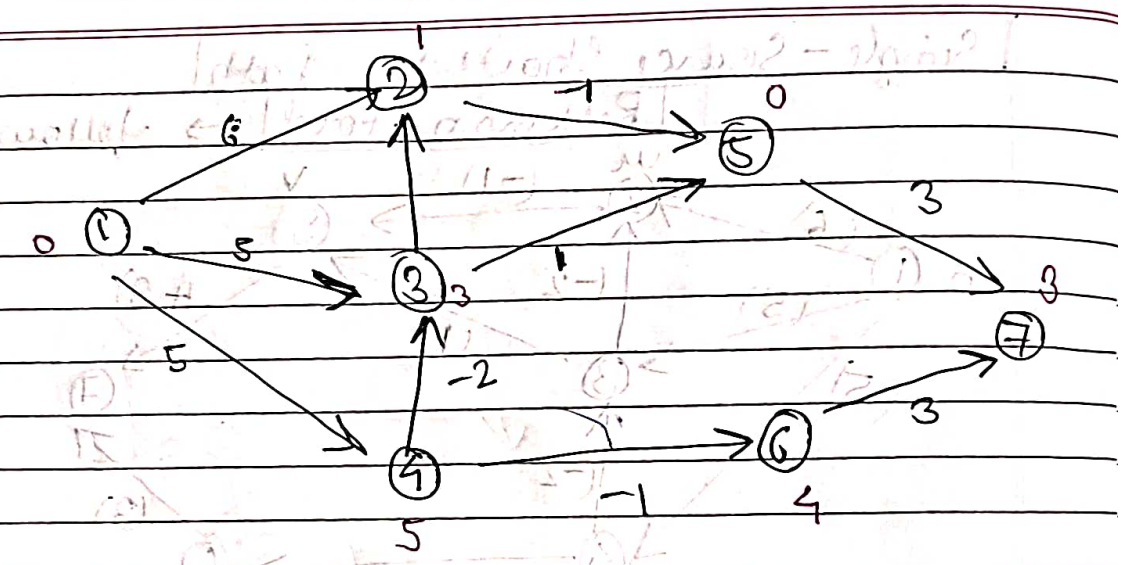
if(d[u] + c(u,v) < d[v])
    d[v] = d[u] + c(u,v).

edgelist: → (1,2), (1,3), (1,4), (2,5), (3,2), (3,5), (4,3), (4,6), (5,7), (6,7).

Step-1) Select Starting vertex as 0, and mark all the rest vertices as ∞.



After 2nd relaxation



Continue upto 6th relaxation, you get the same result.

T-C → Total edges = E

(V-1) times relaxation. So

O(|V||E) → T.C

If V = E = n

then T.C = O(n²)

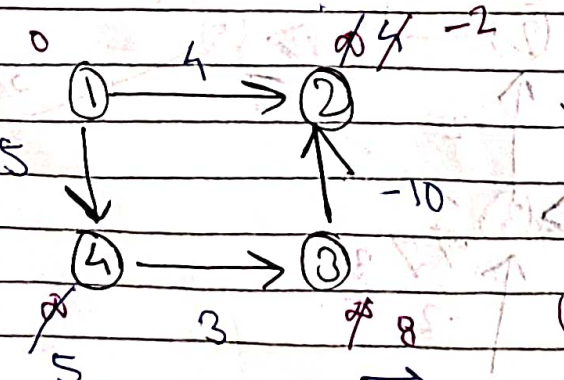
For complete Graph,

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

and they are getting relaxed for (n-1) vertices.

$O\left(\frac{(n-1)\cdot(n-1)\cdot n}{2}\right)$

≈ O(n³)



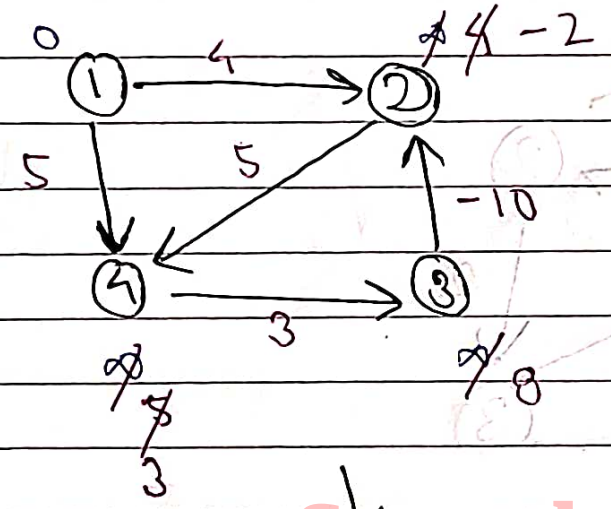
list of edges:
(3,2), (4,3).
(1,4), (1,2)

(4,3) = 3 times relaxation

1 → 0 3 → 8

2 → -2 4 → 5

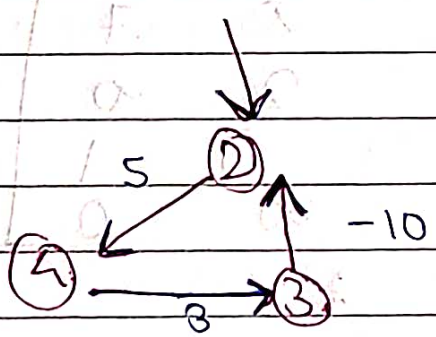
Drawback



→ (n-1) times relaxed

let's relax one more time

vertex 3 will relax from 8 to 6, that shouldn't happen.

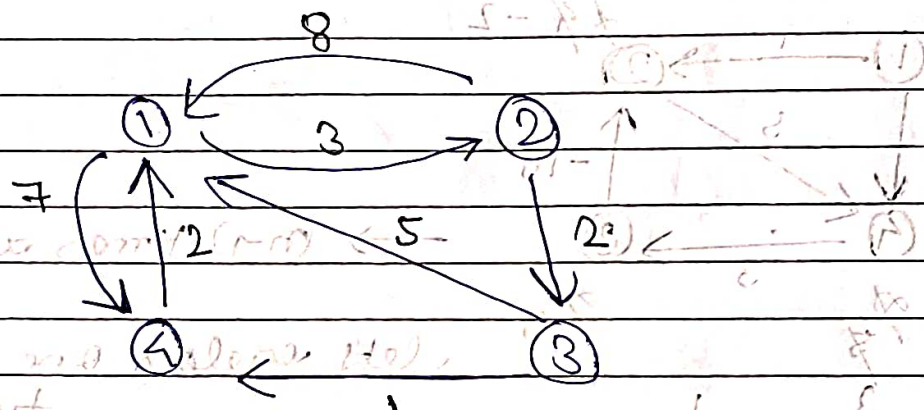


(5+3-10) = -2

(-ve) weight cycle is present.

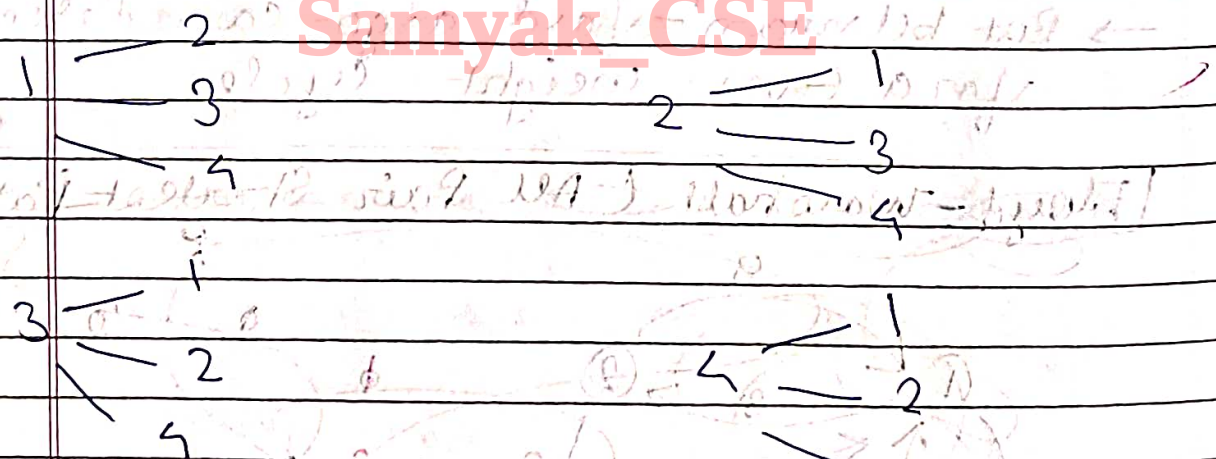
→ But bellman-ford algo- can detect for a (-ve) weight cycle.

Floyd-Warshall Algo
(All Pairs Shortest Path)



$A^0 =$

	1	2	3	4
1	0	3	∞	7
2	8	0	2	∞
3	5	∞	0	1
4	2	∞	∞	0



→ Above problem can be solved by dijkstra's algo, it takes $O(n^2)$, but for 'n' vertex it will take $O(n^3)$ — T.C

$A^1 =$

	1	2	3	4
1	0	3	∞	7
2	8	0	2	15
3	5	8	0	1
4	2	8	∞	0

$A^0[2,3] = 2 < A^0[2,1] + A^0[1,3] = 8 + \infty$

$A^0[2,4]$ $\quad A^0[2,1] + A^0[1,4]$

$= \infty \quad > \quad = 8 + 7$

$A^0[3,2] \quad A^0[3,1] + A^0[1,2]$

$= \infty \quad > \quad = 5 + 3$

$$A^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 5 & 7 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 1 \\ 2 & 8 & 7 & 0 \end{bmatrix} \end{array}$$

$A^1[1,3]$ $\quad A^1[1,2] + A^1[2,3]$

$\infty \quad > \quad = 3 + 5$

$A^1[1,4] \quad A^1[1,2] + A^1[2,4]$

$7 \quad < \quad 3 + 15$

$$A^3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 5 & 6 \\ 7 & 0 & 2 & 3 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix} \end{array}$$

$A^2[1,2]$ $\quad A^2[1,3] + A^2[3,2]$

$3 \quad < \quad 5 + 8$

$$A^4 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 5 & 6 \\ 5 & 0 & 2 & 3 \\ 3 & 6 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix} \end{array}$$

$$A[i,j] = \min \left\{ A^{(k-1)}[i,j], A^{k-1}[i,k] + A^{k-1}[k,j] \right\}$$

$n \times n = n^2$ elements.

```
for (k=1 ; k≤n ; k++) {
    for (i=1 ; i≤n ; i++) {
        for (j=1 ; j≤n ; j++) {
            A[i,j] = min (A[i,j]
                          A[i,k] + A[k,j]);
    }
}
```

For Generating the matrices

→ 3 loops so T.C = $O(n^3)$.

Transitive closure of a Graph

↳ obtained by Warshall's Algorithm.

→ constructed to answer reachability questions.

→ is there a path from a node A to node E in one or more hops?

↳ can determine in O(1) time whether node 'u' is reachable from node v or not.

→ For directed graph $G = (V, E)$, transitive closure of G is a graph $G^* = (V, E^*)$.

- in G^*, there is an edge (u, v) in E^* if and only if there is a valid path from u to v in G.
- $G^* = \{(a,b), (a,c), (a,d), (a,e), (b,c), (b,d), (b,e), (c,d), (c,e), (d,e)\}$

(Q-1) Find Transitive Closure of

$A = \{1, 2, 3, 4\}$

$R = \{(1,1), (1,3), (2,4), (3,3), (4,3)\}$

Write given relation in matrix format,

→

$M_R = W_0$

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

1st Column, 1st row (order is important).

1st Column: 1
1st Row : 1 3
∴ 1 will be inserted at (1,1) ; (1,3).

W_1 =

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

2nd Column: - -
2nd row : - 4
here no - combinations
So

$W_2 = W_1$ only.

3rd Column: 1 3 4
3rd Row: 3
1 will be inserted at (1,3), (3,3), (4,3)

W_3 =

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

4th Column : 2
4th Row : 3
1 is inserted at (2,3)

A_4 =

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

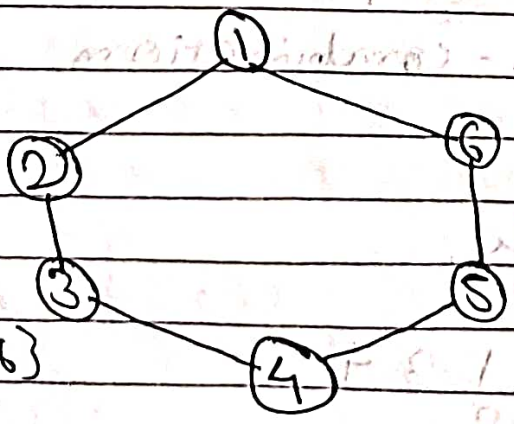
∴ Transitive Closure =

{(1,1), (1,3) (2,3), (2,4), (3,3), (4,3)}

→ Basically transitive closure of a directed graph is a matrix that indicates, whether there exists a path from one vertex to another.

Minimum Cost Spanning Tree

: It is a subgraph of a graph (subset of all vertices) but only (N-1) edges {Bcoz a tree cannot have a cycle}



G=(V,E)

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

E = { (1,2) (2,3) (3,4) ... }

$S \subseteq G$

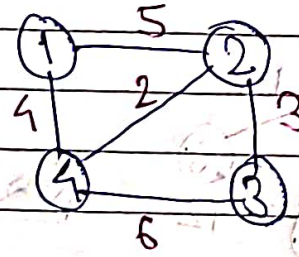
$S = (V', E')$

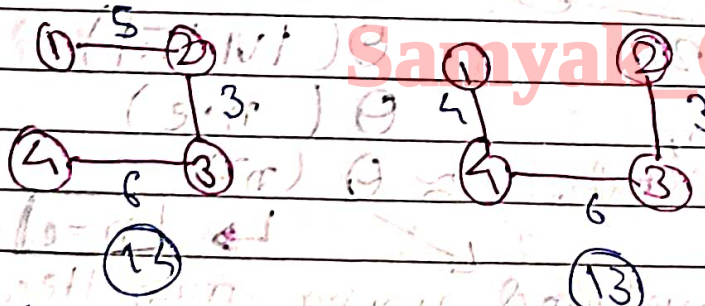
$V' = V$

$|E| = |V| - 1$

~~From a graph, E (E-1) no. of spanning trees can be generated.~~

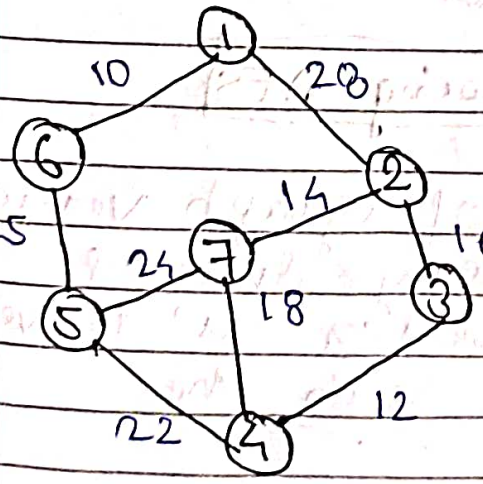
From a graph, ${}^{|E|}C_{|V|-1} - (\text{no. of cycles})$ no. of spanning trees can be generated.

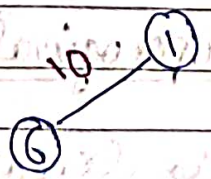




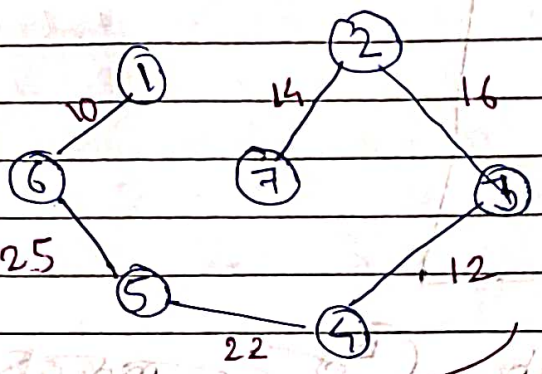
Prim's Algorithm

S-1) Take any min. edge





S-2) now, we cannot take (3,4) bcoz, Prim's algo says that you can only select next min. edge whichever is already connected to (S-1) edges.



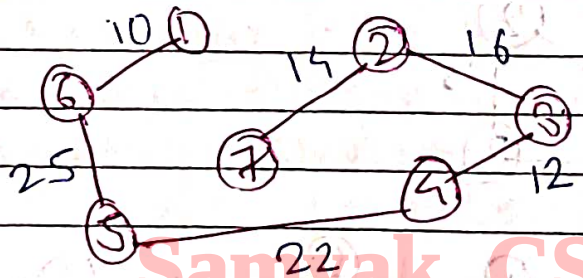
M.S.T

→ = 10+25+22+12+14+16

= 99

Graph must be connected for MST to be made using Prims.

Kruskal's Algorithm



Cost = 99

→ Out of |E| edges, (|V|-1) is selected.

$O(|V||E|)$

$\Theta(n \cdot e)$

$\simeq \Theta(n^2)$

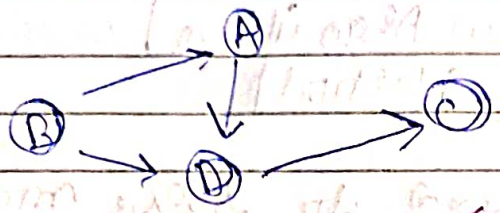
$n = e$

This can be reduced using min. Heap Data Structure.

→ T.C = $O(n \log n)$

Topological Sort using DFS

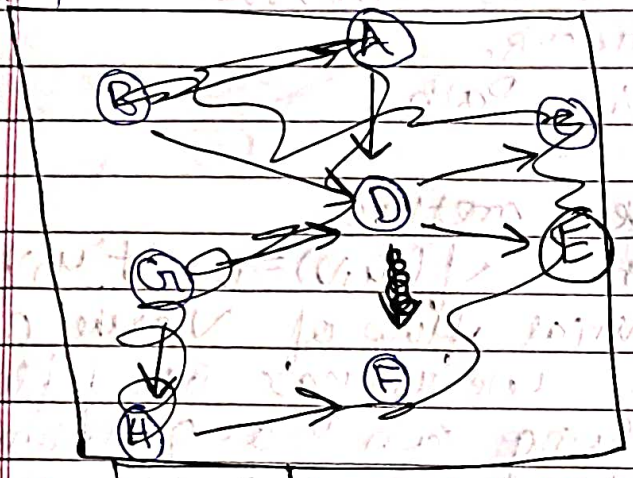
- It is linear ordering of graph vertices such that for every directed edge uv from vertex 'u' to vertex 'v', u comes before v in the ordering.
- Applicable on Directed Acyclic Graph.
- T.C is linear.
- It is not unique.

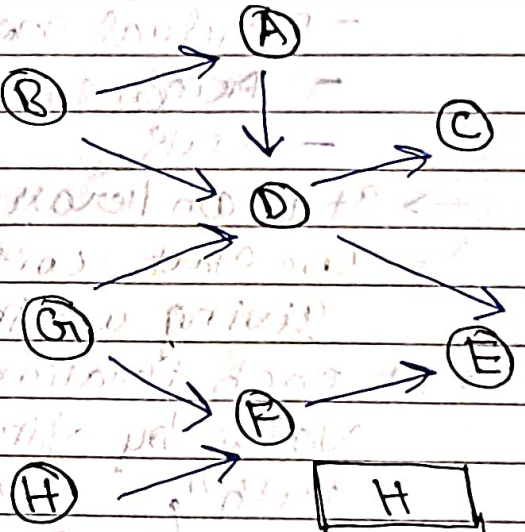


Topological order:

BADC, ABCD (let) ✓ ✗

ex-2)

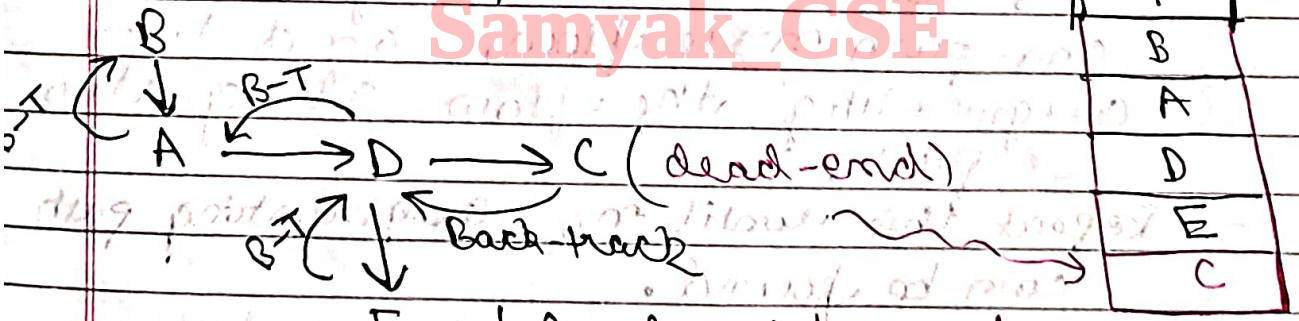




dfs(B)

So V = B

V = visited,



(A Stack with size = no. of nodes of tree).

now, dfs(G) → B-T

↓

F (dead-end)

dfs(H)

now according to LIFO-principle,

pop all the elements

H, G, F, B, A, D, E, C. (Valid Topological order after Sorting)

T.C = O(V+E)

≈ O(linear)

S.C = O(V) (Because of stack only).

Network Flow Algorithm

Ford:- Fulkerson Method

* Above method is used to solve max. flow problems.

→ Based on three basic methods:-
- Residual network.
- Augmenting Path.
- Cut.

→ It is an iterative method.
We start with $f(u,v) = 0 \ \forall \ u,v \in V$ giving an initial flow of value 0.

→ At each iteration, we increase the flow value by finding an "augmenting Path", which can we think of simply as a path from the source 's' to the sink 't' along which we can push more flow, and then augmenting the flow along this path.

→ Repeat this until no augmenting path can be found.

Residual Network

Given $G(V,E)$ with s, t
So C_f (capacity flow) $= C(u,v) - f(u,v)$

Ex) If max. capacity = 10
Bottle Neck = 4
then $C_f = (10-4) = 6$

Augmenting Path

Alternative path from (s) to 't':

$C(p) = \min \{ C_f(u,v) : (u,v) \text{ is in } p \}$

Cuts of flow network

6

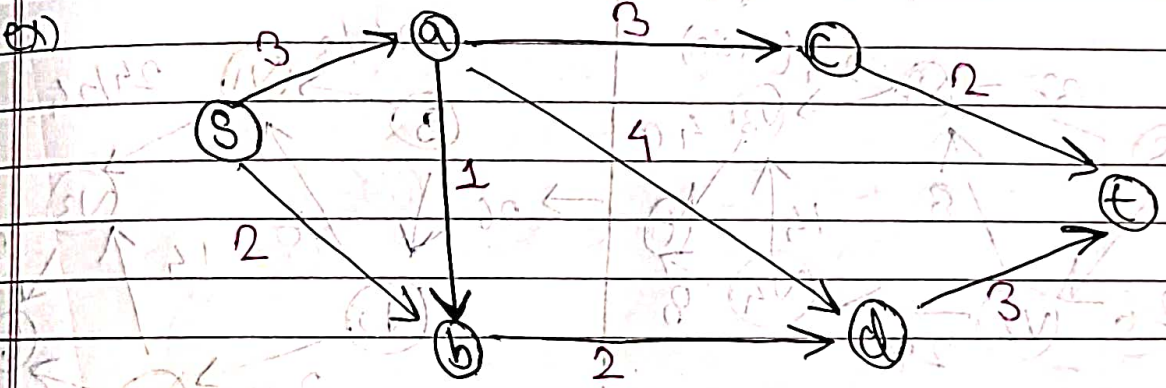
4

(directed opposite

max. cap = 10, B-N = 4

Cuts = 6,

(cannot be re-used

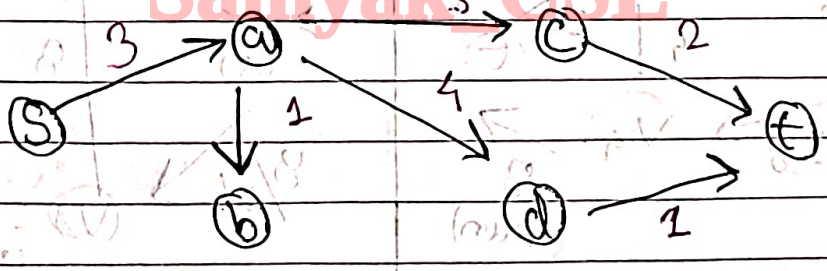


① Choose path S, b, d, t

Min(2, 2, 3) = 2

So B-N = 2

② Add it to flow.

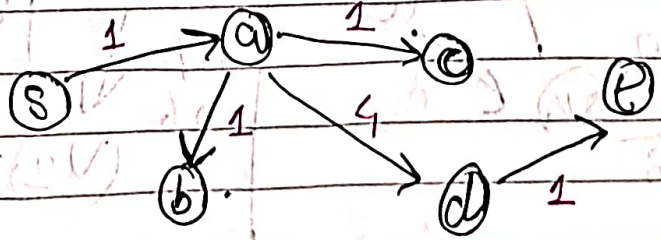


③ Find, other possible Path.

④ Got S, a, c, t

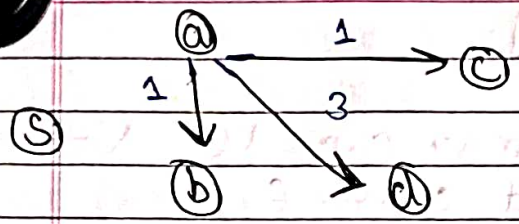
B-N = 2

Add it to the flow.



⑤ New Path, S, a, d, t B.N = 1.

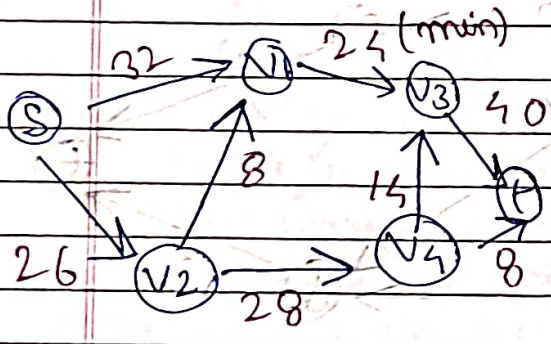
Add, it to the flow

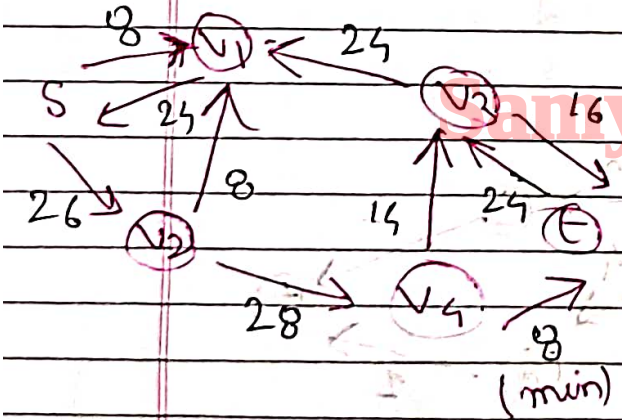


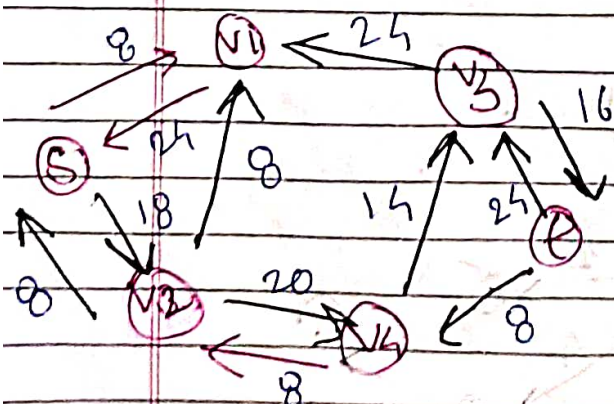
→ now, no path is left.

So max flow = (2 + 2 + 1) = 5 { Σ(B·N) }

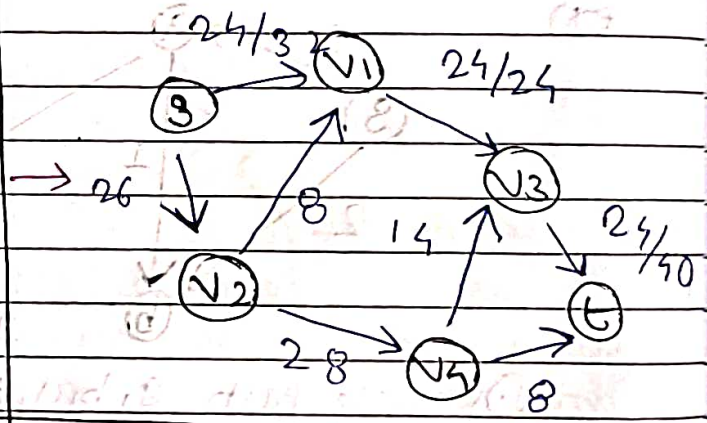
Ex) Residual network and augmenting path

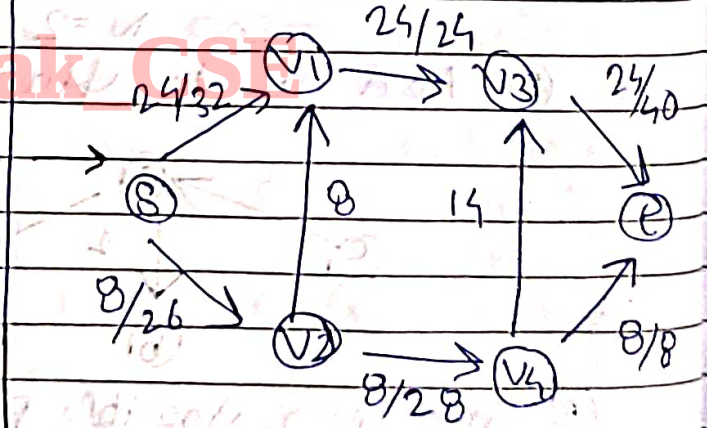


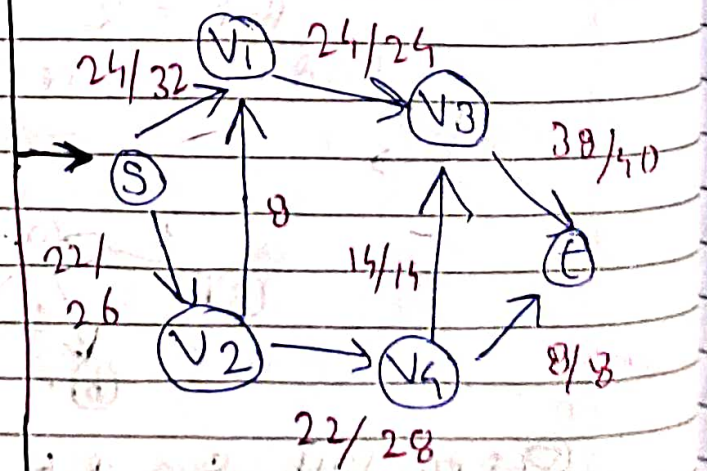


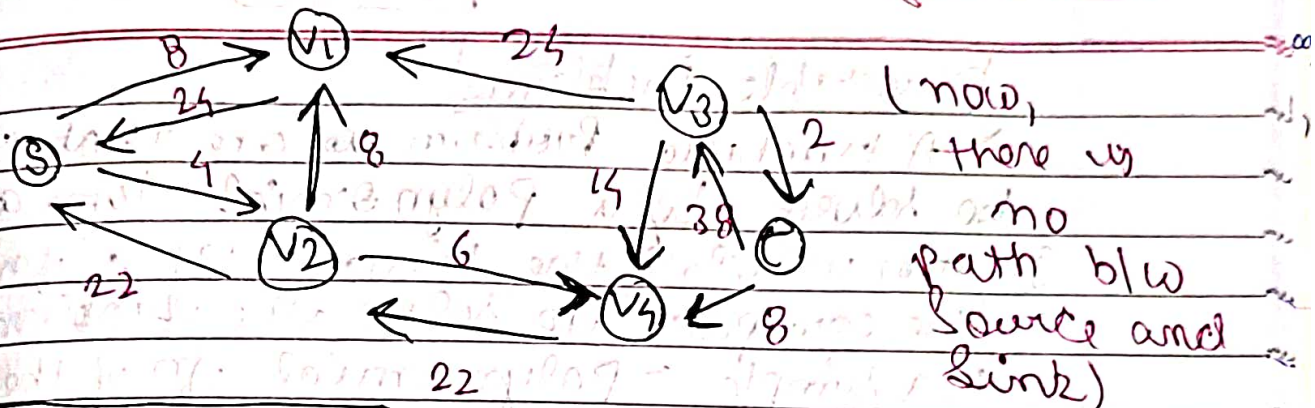


Max - Flow Network





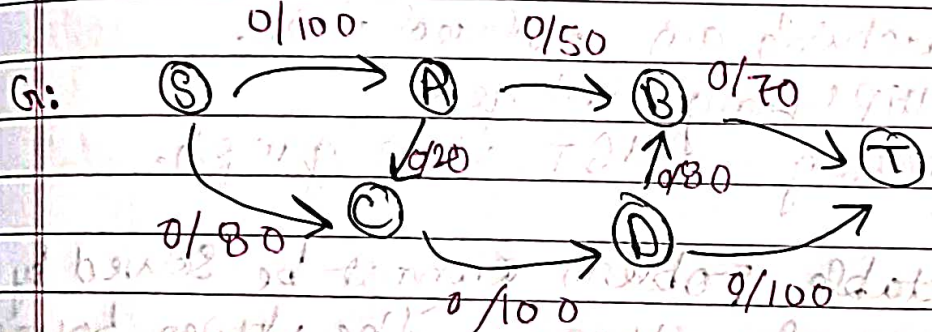


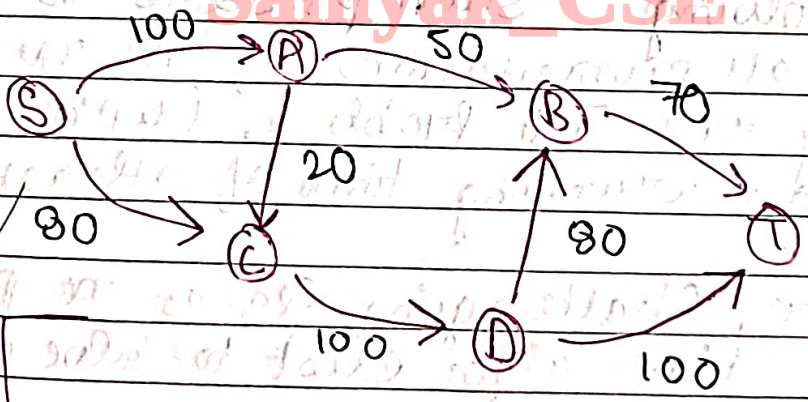


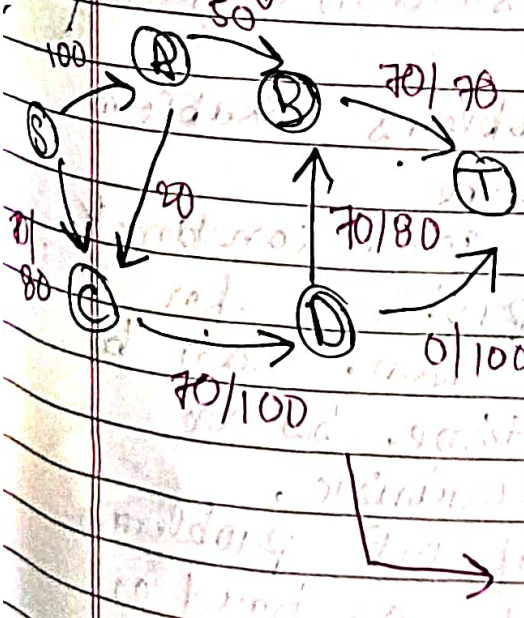
(now, there is no path b/w source and sink)

Ex)

Input GiST







Augmenting Path	B-N Capacity
P1: S → C - D - B - T	70
P2: S - C - D - T	10
P3: S - A - B - D - T	50
P4: S - A - C - D - T	20

Network flow algo:-
↳ Revise

classmate
Date
Page

Tractable Problems

→ A tractable Problem is one that can be solved by a polynomial-time algo. In other words, the time reqd. for the computer to solve the problem is a simple polynomial fn of the problems input size.

ex:-

- Searching an unordered list.
- Searching an ordered list.
- Multiplying integers.
- Finding MST in a graph.

→ An intractable problem cannot be solved by a polynomial-time algorithm. The lower bound for solving such problems is exponential.

ex) Listing all permutations of 'n' numbers.

- Solving the TOH problem. (with a worst-case running time of atleast $(2^n - 1)$.

→ They are challenging Qs as no-polynomial time algo. exist to solve them efficiently.

Computability of Algorithms: (Based on difficulty)

- P - Class → contains problems solvable in polynomial time
  ex) → $O(n^k)$ for some constant k.
- NP Class → contains problems for which a ~~problem~~ soln. can be verified in polynomial time, but are non-deterministic.
- NP-Complete → A subset of NP problems that are at least as hard as the hardest problems in NP
- NP-Hard → Problems that are at least as hard as NP-Complete but may

not be in NP.

NP-Hard and NP-Complete

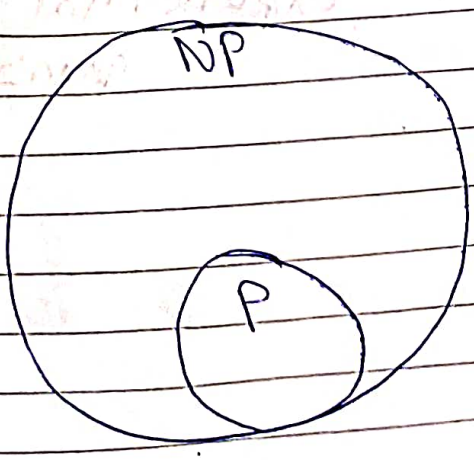
Polynomial Time	Exponential Time
Linear Search – n	0/1 Knapsack – 2^n
Binary Search – $\log n$	TSP – 2^n
Insertion Sort – n^2	Sum of Subsets – 2^n
Merge Sort – $n \log n$	Graph Colouring – 2^n
Matrix Multiplication – n^3	Hamiltonian Cycle – 2^n

Non-deterministic Algo:-

```
Algorithm NSearch (A, n, key)
{
    j = choice ();        → non-deterministic
    if (key = A[j])          statements
    {
        write (j);
        Success ();       →  " " "
    }
    write (0);
    Failure ();           →  " " "
}
```

O(1)

→ Today, this above algo. doesn't works, but, maybe in future it will be made into action, so that above algo. can work in O(1) T.C.



(exponential time problems)

→ We won't work on all problems one-by one but we will try to relate all those problems by a single algo. So that our motive to find a polynomial T.C algo for these exponential T.C problems is accomplished.

→ To relate them, we need some base problem, so we take, Satisfiability problems from propositional calculus. (Part of discrete mathematics).

CNF - Satisfiability Problem

$x_i = \{x_1, x_2, x_3\}$

CNF = $\underbrace{(x_1 \vee \bar{x}_2 \vee x_3)}_{C_1} \wedge \underbrace{(\bar{x}_1 \vee x_2 \vee \bar{x}_3)}_{C_2}$ (Formula)

C_1, C_2 are clauses formed by disjunction and they're joined by conjunction.

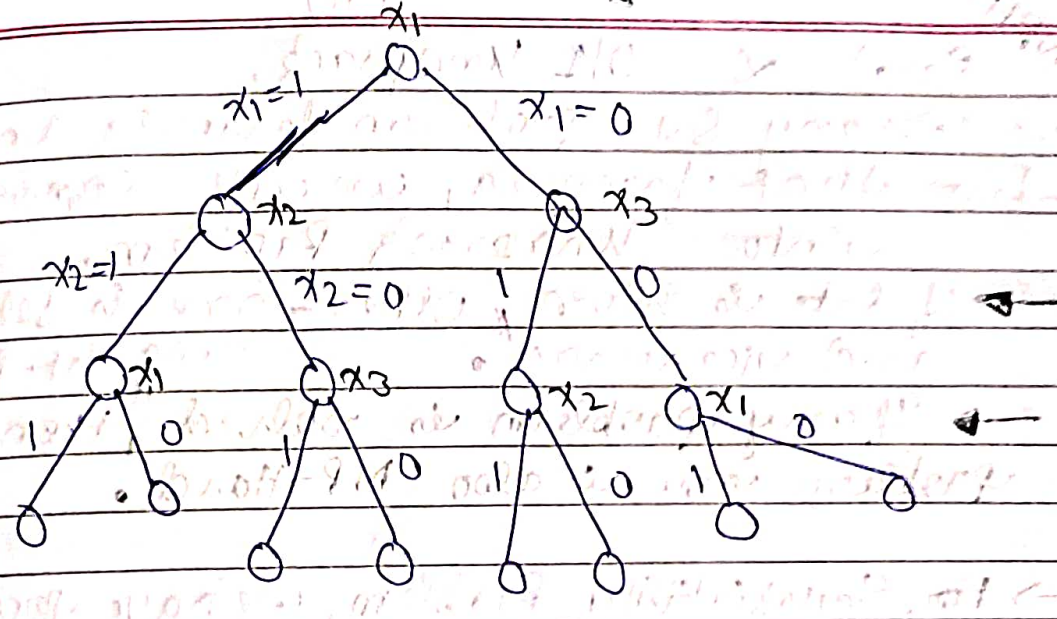
For what value of x_i, CNF formula is true?

x_1	x_2	x_3
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

$i = 2^3$

For n variables, 2^n → T.C

So this problem is similar to a exponential T.C problem.



State-Space Tree the root

→ Obviously like branch-bound algo. Here gives the Soln.

CNF Satisfiability Prob → 2^n

0/1 knapsack → 2^n

Ex) $P = \{10, 8, 12\}$ $n = 3$

$W = \{5, 4, 2\}$ $m = 8$ → capacity of bag

Soln = $x_i = \{ \underline{0/1}, \underline{0/1}, \underline{0/1} \}$

For 3 objects → 0/1

So total $2^3 = 8$ Possibilities.

So 0/1 knapsack has also same State-Space Tree.

By this both Problems can be related.

NP Hard

For time being, let's call Satisfiability problem as NP-Hard, and, rest other exponential Qs as Hard.

→ To show similarity between them, we have reduction technique.

convert in polyn. time

(NP-H) Sat. ∝ 0/1 knapsack. (H)

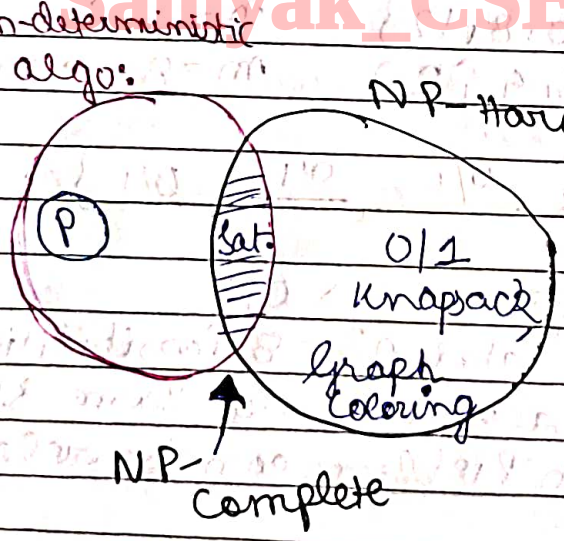
→ We take any Sat. problem formula and from that formula, we will convert into knapsack Problem.

→ If Sat is solved, other one is solved and vice-versa. using Sat problem

→ If any problem is solved, then that problem now is also NP-Hard.

→ For, Satisfiability Problem, we have proved it is NP-Hard, as well as NP-Complete. and we have a Polyn. T.C algo. to Solve Satisfiability Problem.

→ If any Problem has non-deterministic time algo, then that problem is NP-Complete.



So, $P \subseteq NP$

→ Actual Posn. of P is unknown bcoz, as day-by-day our knowledge is expanding, P will become part of deterministic algo.

→ Cook's said that Satisfiability Problem is in P iff and only iff P is in NP.

→ Cook's theorem

Cooks Theorem

→ Satisfiability Problem is an NP-Complete Problem. meaning, SAT Problem is both in the class NP and hard enough that any problem in NP can be reduced to SAT, in Poly. T.C by a Deterministic Turing Machine.

Standard NP-Complete Problems

- SAT Problem.
- Clique Problem → Given an undirected graph G and an integer 'k', determine if there exists a clique (complete subgraph) of size k in G.
- Subset-Sum Problem.
- TSP
- Hamiltonian Cycle Problem.

Reduction Techniques → From notes

Randomized Algorithms

→ An algorithm that uses a random no. at least once during the computation to make a decision.
→ Follows probabilistic analysis.
→ Uses random no. generator.

eg → Picking an edge from a graph randomly.

→ Here, execution time and output varies for the same input given at various instances.

Pro's

- Simple implementation.
- Fast
- Produces optimum output with very high probability.

Con's

- Algo. will not always reach the global optimum soln.
- Sometimes, the sub-optimal soln. is better than the final soln.

Approximation Algorithm

- also called as heuristic algo.
- Way of approaching NP-Completeness for the optimization problem.
- To design an algo. and come as close as possible to the optimum value in a reasonable amount of time.
  (atmost polynomial time).

ex) For TSP, the optimization problem is to find the shortest cycle, and the approximation problem, is to find a short cycle.

ex) Bin Packing, Knapsack Problem, TSP, Vertex Cover.

→ An algo. 'A' is said to be an approx. algorithm for P(optimization problem) if for every given instance I, it returns an approximate soln.

(Polyn. Space)

① P-Space → It consists of problems that can be solved using an algo. with space usage poly. in the input size.

- Unlike NP, PSpace focuses on space constraints
- Decision problems in PSpace are solvable in Polynomial space.

ex. of PSpace problems include →
- Solving Puzzles like 8/15 Puzzle
- Quantified SAT
- Players name capital cities based on specific rules.

② NP ⊆ PSpace

- Any problem in NP can be solved using Polynomial space.
- We can reduce an NP Problem to 3-SAT (a PSpace - complete problem) and then solve it using a polyn. no. of calls to 3-SAT.