End Sem 2016-17 Even Sem

Ans: 5

(a)

Polynomial time	Exponential time / Intractable problem
eg. Linear Search	0/1 knapsack problem
Binary Search	Travelling Salesman
Insertion Sort	Graph Coloring
Merge Sort	

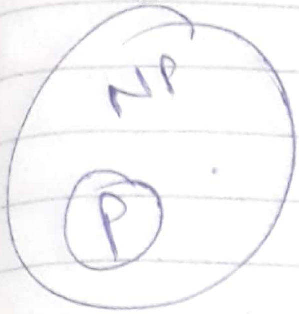
Class P problems → Problems which are solvable in polynomial time (which are tractable).

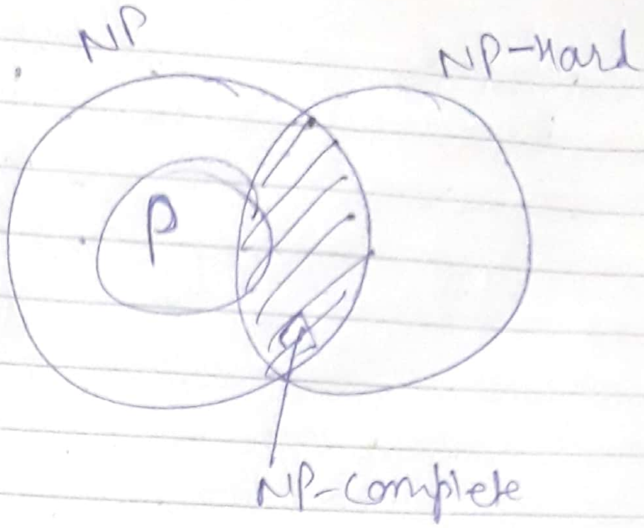
NP Problems: If we don't have a deterministic polynomial solution then we can write a non-deterministic ~~po~~ but polynomial solution. Some statements we write simply which can be solved in future & we assume that they will take O(1) time. All these problems belongs to NP Problem. Problem which can't be solved in polynomial time but is verified in polynomial time.

NP- Complete Problems: We have two types of NP Problems i) NP-Hard & ii) NP Complete

NP-hard: The problems which are not solvable in polynomial time. These are the hardest problems.

If we write some non deterministic polynomial solution for a NP-hard problem then it becomes NP-Complete Problem





First problem identified as NP-~~hard~~ problem is satisfiability problem.

If somehow satisfiability is ~~reduced~~ in ~~to~~ P then P = NP

(b) Steps in proving a problem to be NP-Complete:
Let problem is L.

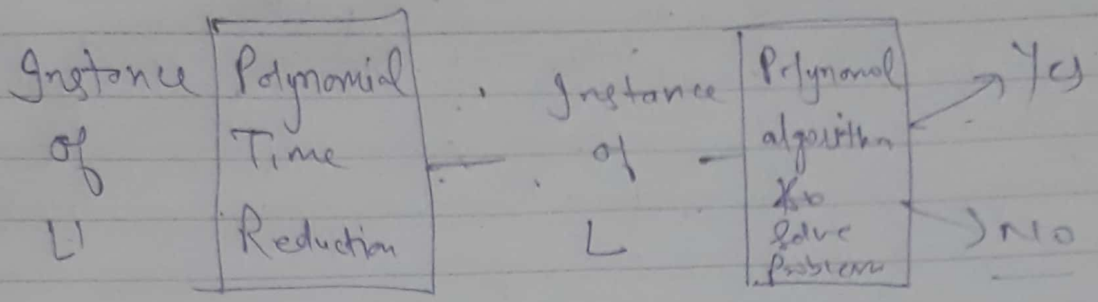
1) Prove that Problem $L \in NP$ (that is that given a solution we can verify it in polynomial time).

2) Select a known NP-Complete problem L'.

3) Describe an algorithm f that transform L' into L in polynomial time.

4) Prove that algorithm is correct ($x \in L'$ iff $f(x) \in L$)

5) Prove that algo f runs in polynomial time.



(C) 3 CNF-SAT (3 Conjuctive Normal form Satisfiability)

The 3SAT Problem is one of the most common NPC problems. ~~used~~

e.g. $F = \underset{c_1}{(x_1 \vee x_2)} \wedge \underset{c_2}{(\bar{x}_1 \vee \bar{x}_2)} \wedge \underset{c_3}{(x_1 \vee x_3)}$ etc

we have three clauses

~~so is~~ 3CNF

Vertex Cover Problem: Let $G^{(V,E)}$ be an undirected graph. A vertex cover of G is a subset of V such that for every $(u, v) \in E$, at least one of u or v $\in$ Cover.

VC – vertex Cover Problem

Instance: a graph $G = (V, E)$ and a +ve integer $k \leq V$

Problem: Is there a subset COVER $\subseteq$ V of size k s.t. each edge in E has at least one endpoint in COVER?

Reduction of 3-SAT to VC.

The 3-SAT is used on the left side of polynomial time reduction
The transformation involves taking a boolean formula that would be "yes" instance to 3-SAT and converting each clause to a set of nodes and edges that are used as an instance of the VC problem.

Showing VC is NPC

1) Show VC is NP

Given an instance and certificate, the validation require checking the ends of each edge to see if one end is in cover. In an n node graph, there are $O(n^2)$ edges, so this checking can be done in P time in the no. of edges.

2) Show VC is NP-hard.

Given an instance C of a 3CNF formula (clauses and variables), construct a graph G and the integer k such that G has a set cover of size k iff C is satisfiable

Since VC is NP & NP-hard So it is NP-complete

Since ~~clique~~ satisfiability problem reduced to clique problem. using

$f = (x_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{x}_2) \wedge (x_1 \vee x_2)$
$\quad\quad c_1 \quad\quad\quad c_2 \quad\quad\quad c_3$

$F = \bigwedge_{i=1}^{k} C_i$

we can have a graph

as ~~G=~~ $V = \{ \langle a, i \rangle \mid a \in C_i \}$

$E = \{ (\langle a, i \rangle, \langle b, j \rangle) \mid b \neq \bar{a} \;\&\; i \neq j \}$

So we can use this clique problem & can reduce to vertex cover problem

It means, given an instance / of clique, we will produce a graph $G(V, E)$ and an integer k s.t. G has a maximum clique of k iff / in $\bar{G}(V, \bar{E})$ has a vertex cover of size $= |V| - k$

$|V'| = k$

$(a, b) \in E \Rightarrow (a, b) \notin \bar{E}$

if $(a, b) \in \bar{E}$

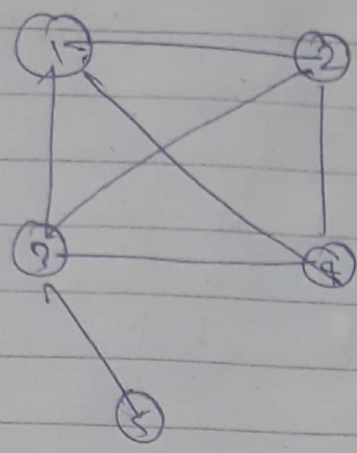
V' is set of vertices of clique size k

Every pair in V' is connected by an edge in E

$\Rightarrow$ at least one of a or b is in $V - V'$

$\Rightarrow$ Edge (a, b) is covered by $V - V'$

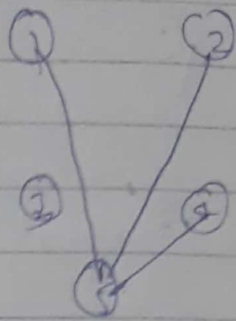
e.g



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

$V' = \{1,2,3,4\}$

$V - V' = \{5\}$



Q.4

(a) DFS(Depth first Search):-

It is recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible else by backtracking.

Backtrack means that when we are moving forward and there are no more along the current path, we move backwards on the same path to find nodes to traverse. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected.

It can be implemented using stack as follows:

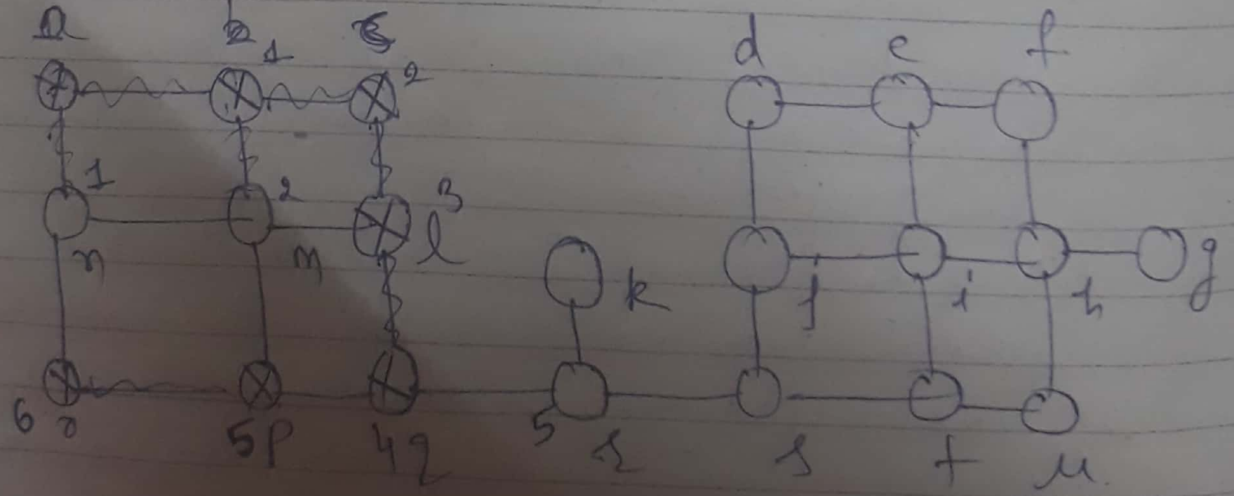
Pick a starting node and push all its adjacent nodes into the stack.
Pop a node from stack and select the next node to visit & push all its adjacent nodes into stack.
Repeat this process until stack is empty.

Pseudocode:

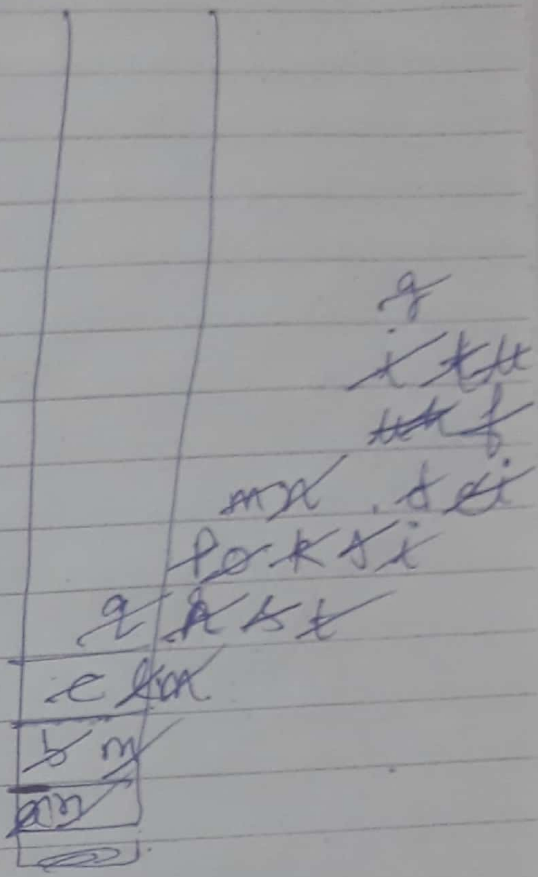
DFS-recursive (G, s):
    mark s as visited
        for all neighbours w of s in graph G:
            if w is not visited
                DFS-recursive(G, w)

Time Complexity O(V+E) when implemented using adjacency list.

(b)



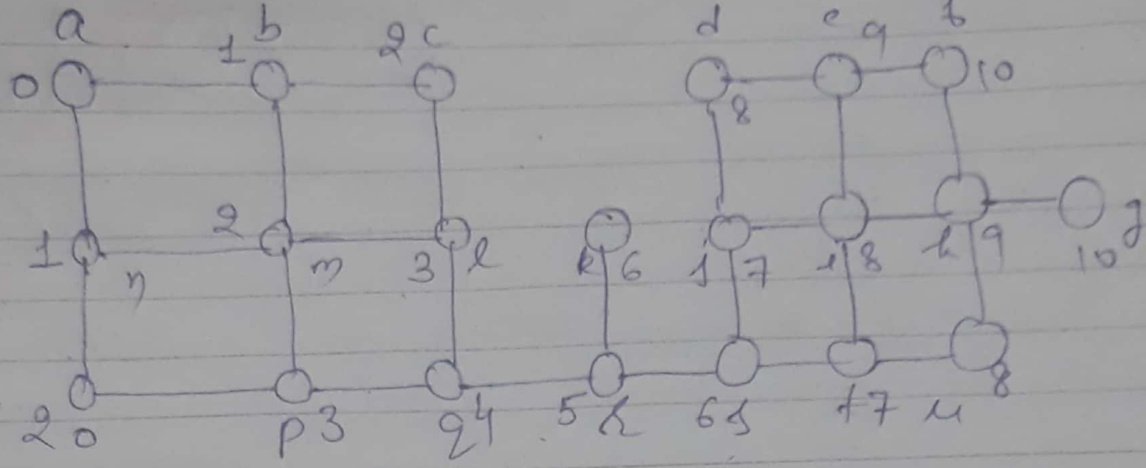
~~a b c l q r p s~~ ... j d e f h g i ~~t~~



	d[v]	f[v]
a	0	NULL
b	1	a
c	2	b
d	8	j
e	9	d
f	10	e
g	12	h
h	11	f
i	~~8~~ ~~10~~ 12	~~f~~ ~~f~~ h
j	7	s
k	6	r
l	3	c
m	~~2~~ ~~4~~ 6	~~b~~ ~~q~~ p
n	~~1~~ 7	~~a~~ m
o	6	p
p	5	r
q	4	l
r	5	q
s	6	r
t	~~7~~ 13	~~s~~ i
u	~~12~~ 14	~~s~~ t

(b) 8

(d)



BFS

queue

~~a~~	~~b~~	~~n~~	~~c~~	~~m~~	~~o~~	~~l~~	~~p~~	~~q~~	~~r~~	~~k~~	~~s~~	~~j~~	~~t~~	~~d~~	~~i~~	~~u~~	~~e~~	~~h~~	~~f~~

a b n c m o l p q r k s j t d i u e h f g

Ans: 3

Principle of optimality in Longest Common Subsequence (LCS):

If $X = \langle x_1, x_2, \ldots, x_m \rangle$ and

$Y = \langle y_1, y_2, \ldots y_n \rangle$ are sequences

let $Z = \langle z_1, z_2, \ldots z_k \rangle$ be some LCS of X & Y

1. If $X_m = Y_n$ ~~then~~ $Z_k = X_m$ and Z_{k-1} is an LCS of X_{m-1} & Y_{n-1}

Scanned by CamScanner

2) If $X_m \neq Y_n$ then Z is an LCS of X_{m-1} & Y
$Z_k \neq X_m \Rightarrow$

or

If $Z_k \neq Y_n$ an LCS of X & Y_{n-1}

(b)

		C	G	A	T	A	A	T	T	G	A	G	A
	0	0	0	0	0	0	0	0	0	0	0	0	0
G	0	0	1	1	1	1	1	1	1	1	1	1	1
T	0	0	1	1	2	2	2	2	2	2	2	2	2
A	0	0	1	2	2	3	3	3	3	3	3	3	3
G	0	0	1	2	2	3	3	3	3	4	4	4	4
A	0	0	1	2	2	3	4	4	4	4	5	5	5
A	0	0	1	2	2	3	4	4	4	4	5	5	6
G	0	0	1	2	2	3	4	4	4	5	5	6	6

GTAAAG

(c) Main Principle Used in KMP pattern matching:

Given a text[0...n-1] and a pattern[0...m-1], write a function search that prints all occurrences of pat[] in text[]

Scanned by CamScanner

The KMP matching algorithm uses degenerating property (pattern having same sub-pattern appear more than once in the pattern) of the pattern and improves the worst case complexity to $O(n)$. The basic idea behind KMP's algorithm is

Whenever we detect a mismatch (after some matches), we already know some of the characters in the next of the next window. We take advantage of this information to avoid matching the characters that we know will anyway match.

- KMP algorithm preprocesses pat[] and constructs auxiliary lps[] of size m (same as size of pattern) which is used to skip characters while matching.

- name lps indicates longest proper prefix which is also suffix.

To Compute LPS

1. $m \leftarrow$ length Pat[]
2. lps[0] $\leftarrow 0$
3. $k = 0$
4. for $q \leftarrow 1$ to m

~~do while~~ if Pat[k] == pat[q]

lps[q] $\leftarrow k+1$;

$k \leftarrow k+1$

else if $k != 0$

$k \leftarrow$ lps[k-1]

$q \leftarrow q-1$;

5, return lps[ ]

e.g Pat = aabaabaaa

lps.

	0	1	2	3	4	5	6	7	8
Pat	a	a	b	a	a	b	a	a	a
Lps	0	1	0	1	2	3	4	5	2

$lps[0] = 0$

$q = 1, k = 0$
$pat[0] = pat[1]$
$lps[q] = 0+1 = 1$

$q = 2, k = 1$
$pat[1] \neq pat[2]$
$k \leftarrow lps[0]$
$k \leftarrow 0$

$q = 2, k = 0$
$pat[0] \neq pat[2]$
$k = 0$

$q = 3, k = 0$
$pat[0] = pat[3]$
$lps[3] = 1$

$k = 1, q = 4$
$pat[1] = pat[4]$
$lps[4] = 1+1 = 2$

$k = 2, q = 5$
$b = b$
$lps[5] = 2+1 = 3$

$k = 3, q = 6$
$pat[3] = pat[6]$
$lps[6] = 3+1$

$k = 4, q = 7$
$pat[4] = pat[7]$
$lps[7] = 4+1 = 5$

$k = 5, q = 8$
$pat[5] \neq pat[8]$
$k \leftarrow lps[k-1]$
$k \leftarrow 2$

$k = 2, q = 8$
$pat[2] \neq pat[8]$
$k \leftarrow lps[k-1]$
$k \leftarrow 1$

$k = 1, q = 8$
$pat[1] = pat[8]$
$lps[8] = 1+1 = 2$

Ans: 2

(a) Divide & Conquer

1. The divide-and-Conquer paradigm involves three steps at each level of the recursion.

Divide: the problem into a number of sub problems.

Conquer the sub problems by solving them recursively.

Combine the solutions to the sub problems into the solution for the original problem.

2. They call themselves recursively one or more times to deal with closely related sub problems.
3. D&C does more work on subproblems
4. In D&C the sub problems are independent of each other

e.g. Binary Search, Merge Search.

(b) Dynamic Programming.

The development of dynamic Programming algorithm can be broken into a sequence of four steps

(1) characterize the structure of an optimal solution

2) Recursively define the value of the solution

3) Compute the value of an optimal solution

4) Construct an optimal solution from computed information.

- DP is not recursive
- It solves the sub-problem only once & then store it for further use.
- DP sub-problems are not independent

e.g Matrix chain Multiplication

(b) Cutting rod:

We are given a rod of length $n \geq 0$
A rod of length i will be sold for p_i dollars

Problem: given a table of prices p_i determine the maximum revenue r_n obtainable by cutting up the rod & selling the pieces.

BOTTOM-UP-CUT-ROD(P, n)

```
let r[0..n] and s[0..n] be new arrays
r[0] = 0
for j = 1 to n
        q = -∞
        for i = 1 to j
            if q < p[i] + r[j-i]
                q = p[i] + r[j-i]
                s[j] = i
        r[j] = q
return r & s
```

(c)

```
min-DENO(DENOMIN, m, v)
{
    let table[0...v] be new array
    for i = 1 to v
        table[i] = INT_MAX;
    for i = 1 to v
        for j = 0 to m-1
            if (denomin[j] <= i)
                sub-res = table[i - denomin[j]]
            if (sub-res != INT-MAX and sub-res+1 < table[i])
                table[i] ← subres+1
    return table[v]
```

Que: How is clique problem reduced to the vertex cover problem?

Ans

~~Thm~~: If a graph G has a clique of size k then the complement of G has a vertex cover of size n-k, where n is the no. of vertices.

It states that the size of the maximum clique in a graph equal the size of a minimum vertex cover in its complement. This is because a set A of vertices is a clique in a graph G iff its complement $\bar{A}$ is a vertex cover in the complement graph $\bar{G}$.

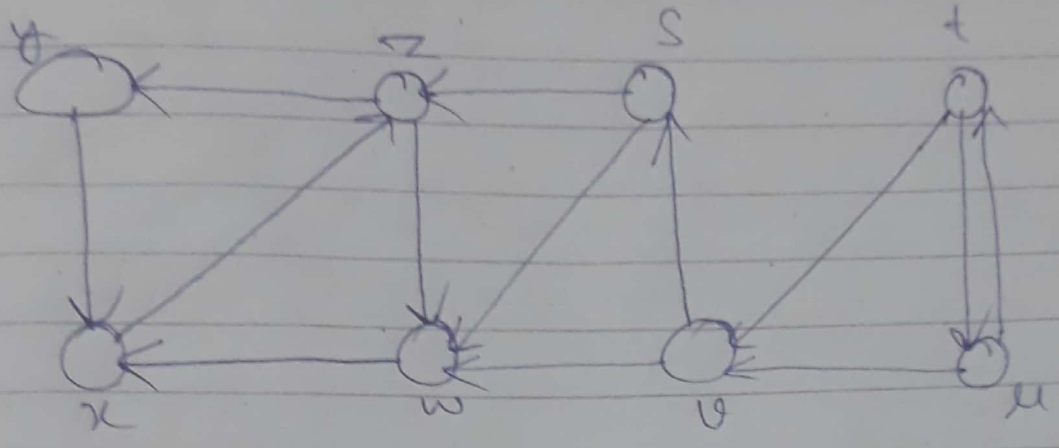
As A is a clique in G so if any two $x, y \in A$ are connected in G; $\bar{A}$ is a vertex cover in $\bar{G}$ if for every edge $(x, y) \in \bar{G}$, one of $x, y \in \bar{A}$.

So $\bar{A}$ is not a vertex cover in $\bar{G}$ if $\exists$ an edge $(x, y) \in \bar{G}$ s.t. $x, y \notin \bar{A}$

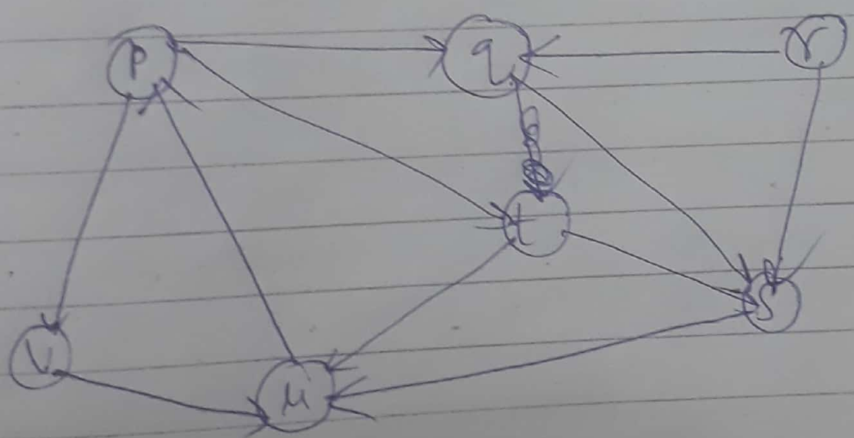
i.e. if for some ~~(x,y)~~ $x, y \in A$, $(x, y) \notin G$

this is exactly the condition that A is not a clique in G.

Dfs Paranthesis



(s (z (y (x x) y) (w w) z) s) (t (v v) (u, u) t)



(p (v (u u) v) (q (s s) q) (t t) p) (r r)