Name : Kamble Sakshi Arun

Roll no : 307A058

Design and Analysis of Algorithms

May June 2022

Date : / /

Page No.: 01

Q.1 a] Consider 0/1 knapsack problem : $N = 3$; $w = (4, 6, 8)$ and $P = (10, 12, 15)$. By using dynamic programming determine the optimal profit for the knapsack of capacity 10.

⟹ We have recurrence formula as

$$V(i,j) = \begin{cases} \max\{V(i-1, j-1), V(i-1, j-w_i) + p_i)\} ; & \text{if } (j-w_i) \geq 0 \\ V[i-1, j) ; & \text{if } (j-w_i) < 0 \end{cases}$$

Item details (p_i, w_i)	i \ j	0	1	2	3	4	5	6	7	8	9	10
[illegible]	0	0	0	0	0	0	0	0	0	0	0	0
(10, 4)	1	0	0	0	0	10	10	10	10	10	10	10
(12, 6)	2	0	0	0	0	10	10	12	12	12	12	22
(15, 8)	3	0	0	0	0	10	10	12	12	15	15	22

$V[3, 10) = \max\{V(2, 9), 15 + V(2, 2)\}$

$= \max\{22, 15 + 0\}$

$= 22$

Thus, the maximum profit of 22 units is earned by selecting items 1 and 2 and the weight of a knapsack is 10 units.

$\therefore x(1;3) = (1, 1, 0)$

b] Explain coin change Making problem in detail?

⟹ Problem description -

1] Let n be the amount for which change is to be made using the minimum number of coins of denominations $d_1 < d_2 < d_3 < \dots d_m$ where $d_1 = 1$

and m is the number of denominations.

2] Assume that there are unlimited coins available for each of the m denominations.

3] Let $C(n)$ be the minimum number of coins whose values sum up to n and $C(0) = 0$. By adding a coin of denomination d_i, $1 \leq i \leq m$, to the amount $n - d_i$ such that $n \geq d_i$ the specified amount n can be obtained.

General procedure -

1] The making change problem computes all denominations whose values sum up to amount n and chooses the one that minimizes $C(n-d_i)+1$.

2] Thus, recurrence formula for $C(n)$ is

$$C(n) = \begin{cases} \min_{i;\, n \geq d_i} \{C(n-d_i)+1\} & ; \text{if } n > 0 \\ 0 & ; n = 0 \end{cases}$$

3] A one-row table is filled from left to right by calculating subproblems the minimum of up to m numbers where m is the number of denominations.

	0	1	......	n
C(n)	0	...	......	$\min_{i;\, n \geq d_i} \{C(n-d_i)+1\}$

4] By backtracking calculations of $C(n)$ we can identify the denominations that produced a minimum value of $C(n)$. Thus, we can obtain the coins in optimal solution.

2] a] Explain how dynamic programming is used to obtain optimal solution for travelling salesperson problem. Also explain why this technique is not used to solve TSP for large number of cities?

⇒ Dynamic programming is a technique used to find optimal solutions to various problems by breaking them down into smaller subproblems and storing the solutions to these subproblems in a table to avoid redundant calculations. When applied to the Traveling salesperson Problem (TSP), dynamic programming works as:

1] Subproblem Definition :
Dynamic programming breaks this down into subproblems, where you calculate the shortest route for subsets of cities.

2] Recursive Formula :
To solve the TSP for a given subset of cities, you consider each city in the subset as the last city visited. You calculate the shortest route by trying all possible combinations of the last city and remaining city in the subset. This is done recursively and minimum route is stored.

3] Memorization :
~~The~~ To avoid recalculating the same subproblems, dynamic programming uses memorization. The results of each subproblem are stored in a table so that if the same subproblem is encountered again.

4] Build Up :
The final result is obtained by considering all possible starting cities and finding the minimum route

While dynamic programming can provide an optimal solution for TSP, it becomes impractical for a large number of cities due to several reasons :

1] Exponential Time Complexity :
The number of subproblems grows exponentially with number of cities, making it computationally expensive.

2] Memory Requirements -
Storing solutions for all possible subsets of cities in a table requires a large amount of memory, which can be a limitation for a large number of cities.

3] Computational Complexity :
The time required to compute and store solutions for all subproblems also increases exponentially, making it inefficient for large instances.

b] What is dynamic programming? Is this the optimization technique? Give reasons what are its drawbacks?

⇒ Dynamic programming is a powerful optimization technique used to solve complex problems by breaking them down into smaller overlapping subproblems and storing their solutions to avoid redundant calculations.

Here's an overview of dynamic programming and its key characteristics :

1] Overlapping Subproblems :-

Dynamic Programming breaks a problem into smaller subproblems, and these subproblems often overlap, meaning the same subproblem is solved multiple times.

2] Optimal Substructure -

Dynamic programming assumes that an optimal solution to a problem can be constructed from optimal solutions of its smaller subproblems. In other words, it exhibits the principle of optimality.

3] Memoization -

To store and retrieve the solutions to subproblems, dynamic programming often uses techniques like memorization.

Drawbacks of Dynamic Programming :

1] Exponential Time Complexity :-

Dynamic Programming can have high time complexity, particularly when dealing with problems with a large number of overlapping subproblems.

2] Space Complexity :-

Storing solutions for all subproblems can lead to high memory usage, which can be a limitation, especially when dealing with problems with larger number of subproblems.

3] Difficulty in Identifying Subproblems :

Identifying appropriate subproblems and formulating a recursive solution can be challenging.

4] Not Always Applicable :

Problems without the property of overlapping subproblems and optimal substructure may not benefit from this approach.

Q.3] a] Find all possible solutions for 5 queens problem using backtracking.

⟹ Let $x[1:5] = (x_1, x_2, x_3, x_4, x_5)$ be a 5-tuple solution to 5-queens problems where on i^{th} queen Q_i is correctly placed on i^{th} row and j^{th} column of a 5×5 chessboard

1] Solution 1 :

Q_1				
		Q_2		
				Q_3
	Q_4			
			Q_5	

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

2] Solution 2 :

Q_1				
			Q_2	
	Q_3			
				Q_4
		Q_5		

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

3] Solution 3 :

	Q_1			
			Q_2	
Q_3				
				Q_4
		Q_5		

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

4] Solution 4 :

	Q_1			
			Q_2	
Q_3				
		Q_4		
				Q_5

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

5] Solution 5 :

	Q_1			
				Q_2
		Q_3		
Q_4				
			Q_5	

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

6] Solution 6 :

		Q_1		
				Q_2
	Q_3			
			Q_4	
Q_5				

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

b] Current configuration is (7, 5, 3, 1) for 8 queens problem. Find the answer tuple using backward method.

⟹ Let $x[1:8] = (x_1, x_2, \dots x_8)$ be an 8-tuple solution to 8-queens problems. When an i^{th} queen q_i is correctly placed on an i^{th} row and a j^{th} column of an 8 x 8 chessboard, $x[i] = x_i = j$

- The current configuration is $x[1:8] = (7, 5, 3, 1, 0, 0, 0, 0)$ for 8-queens problem.

x	x	x	x	x	x	q_1	x
x	x	x	x	q_2	x	x	x
x	x	q_3	x	x	x	x	x
q_4	x	~~q~~x	x	x	x	x	x
x	x	x		x		x	x
x	x	x		x	x	x	
x		x	x	x		x	
x		x		x		x	x

Invalid positions for further placements w.r.t. configuration (7, 5, 3, 1)

Total valid positions w.r.t. $x[1:8] = (7, 5, 3, 1, 0, 0, 0, 0)$ are
- for q_5 : (5, 4), (5, 6);
- for q_6 : (6, 4), (6, 8);
- for q_7 : (7, 2), (7, 6);
- for q_8 : (8, 2), (8, 4), (8, 6);

The final solution $x[1:8] = (7, 5, 3, 1, 6, 8, 2, 4)$ w.r.t. given configuration is depicted as -

						q1	
				q2			
		q3					
q4							
					q5		
							q6
	q7						
			q8				

Q.4] a] State the principle of backtracking. Explain the constraints used in backtracking with an example.

⟹ Principle of backtracking -

1] In backtracking a solution - tuple is determined by evaluating each possible solution in the solution space one by one.

2] The solution space is represented through the state space tree that follows depth first node generation.

3] The bounding function is applied to check the promising and non-promising nodes.

4] Whenever any node violates the bounding function and shows infeasibility, the algorithm stops pursuing that branch further.

5] It then "backtracks" and explores an alternative branch.

6] By discarding non-promising solutions, back-tracking does fewer trials to determine the solution.

Let's consider the classic example of "N-Queens" problem to explain constraints in backtracking:

Constraints:

1] Row constraint - No two queens can be placed in the same row.

2] Column constraint - No two queens can be placed in the same column.

3] Diagonal constraint - No two queens can share the same diagonal.

Constraints play a crucial role in guiding the backtracking algorithm to explore only those paths that lead to valid solutions, ultimately solving the N-Queens problem.

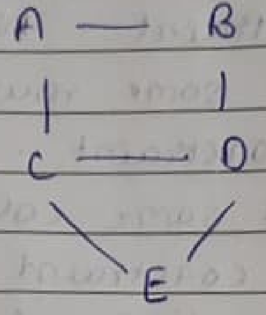
b] What is m colorability optimization problem. Explain with an example.

⇒ It is a graph theory problem that involves assigning colors to the vertices of a graph in such a way that no two adjacent vertices have the same color, using a minimum of 'm' colors. This problem is often referred to as the 'graph coloring problem'.

Problem -

Given a graph G = (V, E) where V is the set of vertices and E is the set of edges. Find the minimum no. of colors (m) required to color the vertices of the graph such that no two adjacent

vertices have same color.
ex - Consider following graph with vertices (A, B, C, D, E) and edges {(A, B), (A, C), (B, D), (C, E), (D, E)}.



To determine the minimum no. of colors (m) needed to color this graph optimally, you would go through the process of assigning colors while ensuring no adjacent vertices share the same color.

For this graph, you can optimally color it with just two colors:
Coloring 1: A, C, E (Red)
Coloring 2: B, D (Blue)

In this class, m=2 and its known as a 2-coloring or a bicoloring. The m-colorability optimization problem aims to find the min value of 'm' that allows for a valid coloring of graph.

Q.5 a) Differentiate between backtracking and branch and bound. Illustrate with example of knapsack problem.

	Backtracking	Branch and Bound
1]	It is generally used to solve non-optimization problems.	It is generally used to solve optimization problems.
2]	A state space tree is generated by using the depth first node generation policy.	A state space tree is generated by using different rules like the best-first rule, BFS, DFS.
3]	All backtracking algorithms search a state space tree by applying DFS only.	Based on different search techniques, B&B algorithms have different variations as LCBB, FIFOBB, LIFOBB
4]	The bounding function is typically a feasibility function that does not make use of the value of best solution seen so far.	The bounding function is heuristic function that computes the bound values at each node and compares them with value of best solution seen so far.
5]	It searches the state space tree until it gets a solution.	It searches the state space tree until it gets an optimal solution.
6]	Eg- N-queens problem, graph coloring, Hamiltonian cycle problem.	Eg - Job sequencing problem, 0/1 knapsack, Travelling salesperson.

Consider 0/1 knapsack problem -

I] By backtracking approach -

1] Start with an empty knapsack.

2] Add items in a specific order (e.g. A, B, C)

3] Check if the knapsack's weight exceeds the capacity.

4] If it exceeds, backtrack and try the next item.

5] If a valid solution is found, compare it with best solution so far.

Branch and Bound -

1] Create initial bounds for the subproblems (e.g. the upper bound is total value of all items, and lower bound is 0).

2] Divide the problem into subproblems, such as including /excluding each item.

3] Compute bounds for each subproblem.

4] Prune branches with bounds indicating they can't improve upon best-known solution.

5] Continue until you've explored all possibilities, updating the best solution as you go.

b] Solve following Job sequencing with deadline problem using Branch and bound.

Job	p	d	t
1	5	1	1
2	10	3	2
3	6	2	1
4	3	1	1

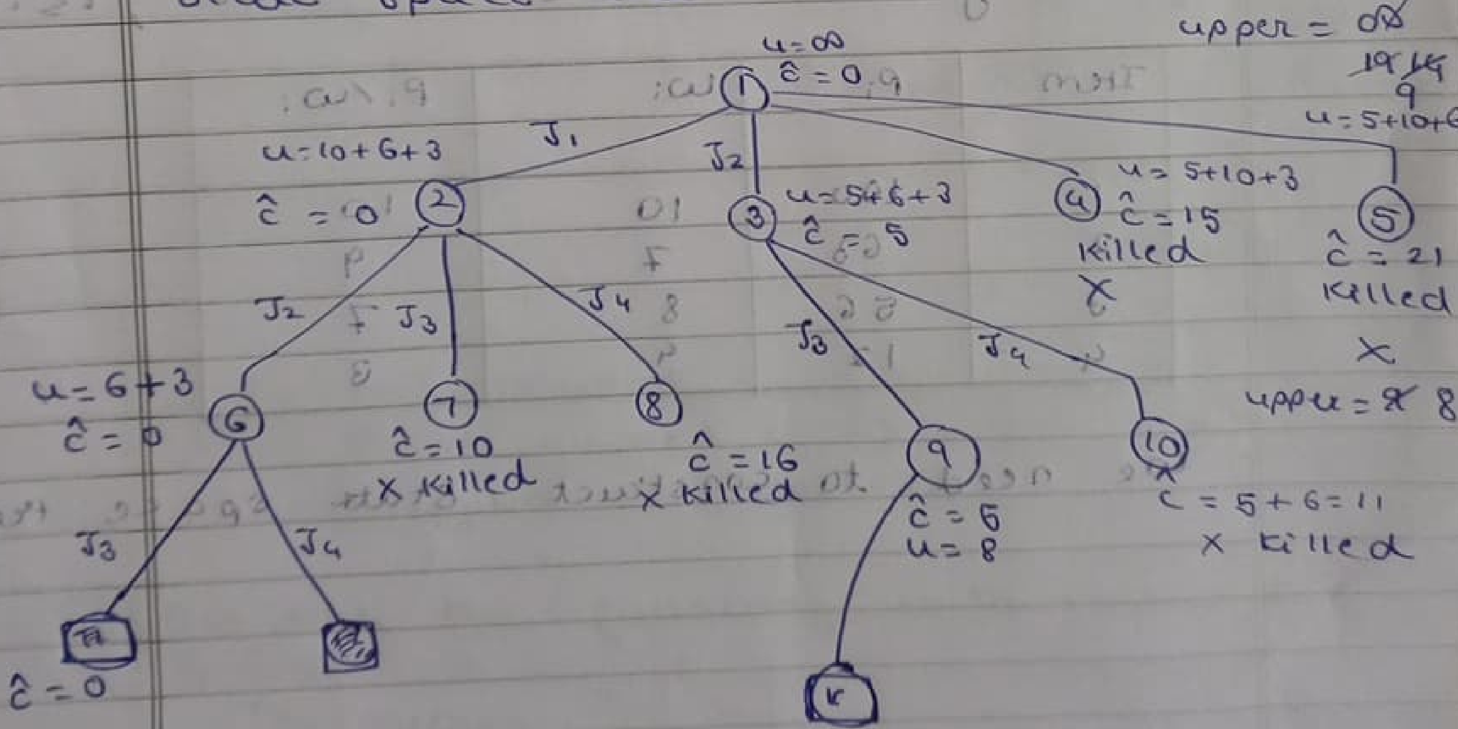
⇒

u = Sum of all penalties except that included in solution.

$\hat{c}$ = Sum of penalties till then last job considered.

$$u = \sum_{i \notin S} p_i \qquad c = \sum_{i \in S_k} p_i$$

State space tree



∴ sequence is J_2, J_3

Profit $= 10 + 6 = 16$

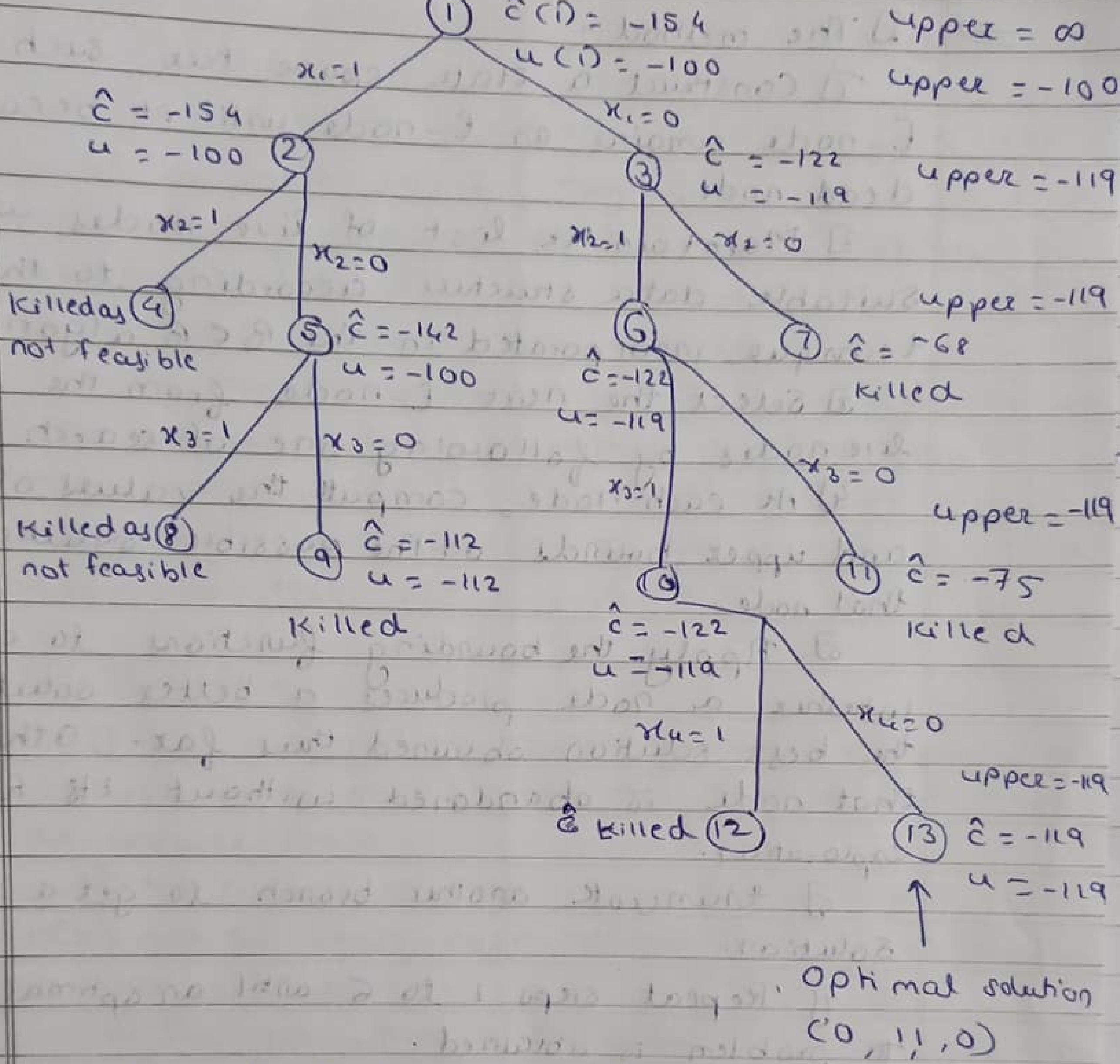
Q.6(a) Solve the following instance of knapsack problem by branch and bound algorithm for $W = 16$.

Item	Weight	value in Rs.
1	10	100
2	7	63
3	8	56
4	4	12

⟹ To solve by LCBB, given items should be in decreasing order of ratio of P_i / w_i, $1 \le i \le n$.

Item	P_i	w_i	P_i / w_i
1	100	10	10
2	63	7	9
3	56	8	7
4	12	4	3

We need to construct state space tree



∴ Optimal solution is (0, 1, 1, 0)
i.e. Profit gained = 63 + 56 = 119
Weights = 7 + 8 = 15

b) Describe the following with respect to B & B
- The method
- LC search
- Control abstraction for LC search
- Bounding function.

⇒ I] The method -

1] Construct a state space tree such that the E-node remains an E-node until it becomes a dead node.

2] Maintain the list of live nodes using a suitable data structure according to the search technique incorporated in the B&B algorithm.

3] Select the next E-node from the list of live nodes by following one of search techniques

4] At each node, compute the values of lower and upper bounds on the possible solutions from that node.

5] Apply the bounding functions to check whether a node produces a better solution than the best solution obtained thus far. Otherwise, that node is abandoned without its further exploration.

6] truncate another branch to get a better solution.

7] Repeat steps 1 to 6 until an optimal solution to problem is obtained.

II] LC search -

- It uses a heuristic cost function to compute the bound values at each node.

- Nodes are added to the list of live nodes as soon as they get generated.

- The node with least value of a cost function is selected as a next E-node.

III] Control abstraction for LC search

- Consider S is a state space tree and $c(*)$ is a cost function in LC search. Let K be a node in S, then $c(K)$ gives the least cost of any answer state in the subtree rooted at node K. So, $c(S)$ can be considered as a cost of least-cost answer state in S.
- As the computation of $c(*)$ is complex, we can replace it by heuristic function as $\hat{c}(\cdot)$ to estimate $c(\cdot)$. $\hat{c}(\cdot)$ should be easily computable and if K is an answer state or a leaf node then $c(*) = \hat{c}(K)$.

IV] Bounding Function -

1] The usage of bounding function prunes the subtrees in a state space tree that do not have an answer state.

2] Each answer state K has an associated cost $c(K)$ and the least-cost answer state is defined as an optimal solution.

3] The estimation of the added cost reaches an answer state from a node K is described by $\hat{c}(K)$ so that $\hat{c}(K) \leq c(K)$. It defines lower bound on solutions feasible from node K.

4] Consider upper gives an upper bound on the cost of a least-cost solution. Then all lives nodes K with $\hat{c}(K) > upper$ can be killed without further exploration since all answer states reachable from node K have cost $c(K) \geq \hat{c}(K) > upper$.

5] Initially $upper = \infty$ and whenever a new answer state is obtained, the value of upper is updated

Q.7 a) When do you claim that algorithm is polynomial time algorithm? Explain with an example.

⇒ Polynomial-time algorithm -

1] It is an algorithm whose running time is polynomially dependent on the input size of a problem instance.

2] Thus, a polynomial-time algorithm for a problem instance of size n has its worst-case complexity $O(p(n))$ where $p(n)$ is polynomial of input size n.

Eg. Time complexities $O(n^2)$, $O(n^3)$, $O(1)$, $O(n \log n)$.

3] P-class problems are decidable & tractable.

4] Examples - linear search ; $O(n)$, binary search : $O(\log n)$, merge sort : $O(n \log n)$; Prim's algorithm : $O(n^2)$; Floyd-Warshall's : $O(n^3)$.

Example - Finding the maximum element in an array.

In this algorithm, the time complexity is $O(n)$, where 'n' is size of input array. The algorithm iterates through the array once, comparing each element with current maximum. Since the time complexity is linear function of input size, it is considered a polynomial time algorithm.

This means that regardless of how large the input array is, the time it takes to find the maximum element grows at most linearly with size of array, making it an efficient algorithm for practical purposes.

b] Explain i] Complexity classes.
ii] Deterministic Algorithms.

⇒ I] Complexity classes -
It describes the categories of computational problems based on their algorithmic complexities.

1] P - class
- It is the class of decision problems that can be solved in polynomial time by deterministic algorithms.
- Polynomial time algorithm is an algorithm whose running time is polynomially dependent on the input size of a problem instance.
- P-class problems are decidable and tractable.

2] NP - class
- It is the class of decision problems that can be solved in polynomial time by non-deterministic algorithms.
- NP stands for Non-deterministic Polynomial time algorithm which produces possible solutions to given problems in a non-deterministic way and verify the correctness of those solutions in polynomial time.

3] NP-Hard Class -

- A decision problem P_1 is NP-Hard if each NP-class problem is polynomially reducible to P_1.

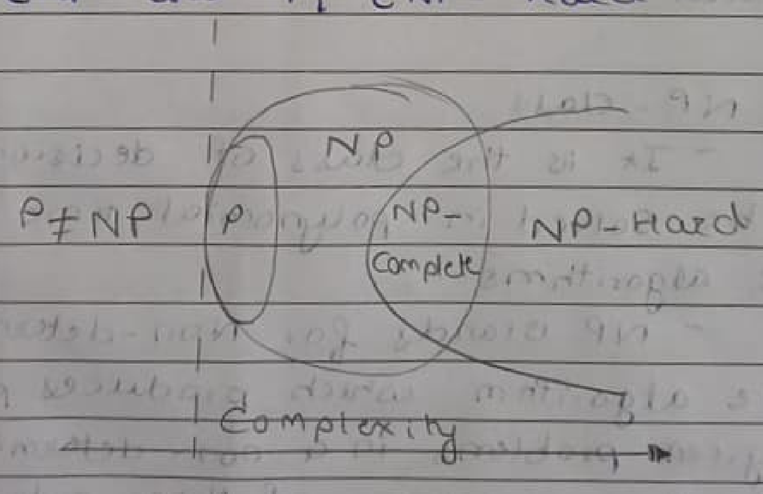
- Thus, it implies that an NP-Hard problem is at least as hard as the hardest NP-Class problem.

- NP-Hard problems are decidable or undecidable.

4] NP-Complete Class

- A decision Problem P_1 is NP-Complete if i] P_1 is an NP-Class problem (i.e. $P_2 \in NP$) and ii] Each NP-class problem is polynomially reducible to P_1. (i.e. for each problem $P_2 \in NP$, $P_2 \propto P_1$).

- Thus P_1 is said to be NP-Complete if $P_1 \in NP$ and $P_1 \in$ NP-Hard



II] Deterministic Algorithms.-

- The algorithm is said to be deterministic if it generates the same result for the same set of inputs.
- The deterministic algorithm uniquely define the outcomes for specific legitimate input.
- All computer programs are deterministic.
- Eg. Addition of first n numbers, sorting algorithms, search algorithms.

Q.8] a] Explain vector cover problem in detail.

⟹ Problem description -

- A vector cover problem has two general variants as below:

i] vertex cover optimization problem

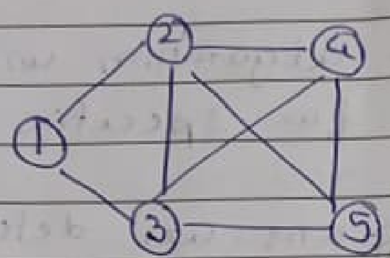
ii] vertex cover decision problem.

- A vertex cover optimization is to determine a vertex cover with minimum number to nodes for a given undirected graph.
- A vertex cover decision problem is to check whether a given undirected graph has a vertex cover of size at most S for some given S.

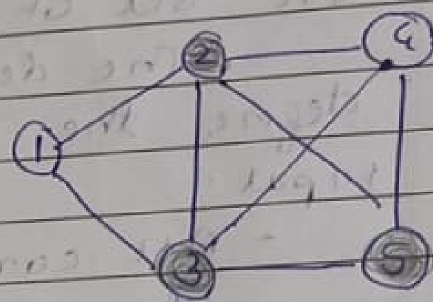
1] A vertex cover of a given undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ iff, each edge $<v_0, v_1> \in E$, $v_0, v_1 \in V_1$ is incident to at least one node in V' that means either $v_0 \in V'$ or $v_1 \in V'$ or $v_0, v_1 \in V'$.

2] Size of vertex cover $(|V'|)$ is no. of vertices in it.

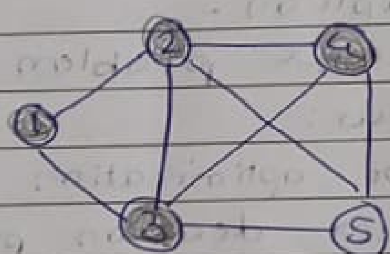
5] Since each node in a vertex cover V' "covers" its incident edges, all nodes in V' covers all edges in E of given graph G = (V, E)



Given graph



Vertex cover of size 3
(smallest vertex cover)



Vertex cover of size 4

6] What is deterministic algorithm? Write any one deterministic algorithm.

⇒ Deterministic algorithm -

- It generates the same result for the same set of inputs.
- It uniquely define the outcomes for specific legitimate input.
- All computer programs are deterministic.

Example - Binary search algorithm.

- It is used to efficiently find a specific element in a sorted array.
- Binary search is a deterministic algorithm because, given a sorted array and a target element, it will always follow the same steps and produce the same result for the same input.
- There is no randomness or non-deterministic behavior involved in its execution.