1. Divide and Conquer: Divide and Conquer is an algorithm design paradigm that works by recursively breaking a problem into smaller sub-problems until the sub-problems are simple enough to solve directly. Then, the solutions of the sub-problems are combined to form a solution to the original problem.
Steps: Divide: Split the problem into smaller sub-problems. **Conquer:** Solve each sub-problem recursively. **Combine:** Merge the solutions of the sub-problems to get the final solution.
Merge Sort Example: Merge sort follows the divide and conquer approach: Divide the array into two halves., Recursively sort each half, Merge the two sorted halves.
Let the array be: [38, 27, 43, 3, 9, 82, 10] • Divide into [38, 27, 43] and [3, 9, 82, 10] • Recursively divide and sort • Merge sorted halves **Time Complexity Derivation:** Let T(n) be the time complexity of merge sort: • Divide: takes O(1) • Conquer: 2 sub-problems of size n/2 ⇒ 2T(n/2) • Combine: merging takes O(n)
So, T(n) = 2T(n/2) + O(n), Using the **Master Theorem**, we get: T(n) = O(n log n)

2. Knapsack Problem, Greedy Method: Greedy algorithm for Knapsack problem selects items based on **maximum profit/weight ratio**. **Example:** Weights = [2, 3, 4, 5], Profits = [1, 2, 5, 6], Capacity = 8
Profit/Weight ratios = [0.5, 0.66, 1.25, 1.2], Pick item 3 → weight 4, profit 5
Pick item 4 → weight 5 (exceeds capacity, pick 4 units if fractional allowed)
Greedy Total Profit (Fractional): Item 3: profit 5, Remaining capacity: 4, Take 4/5 of item 4: (4/5) × 6 = 4.8
Total = 9.8 **0/1 Knapsack (Optimal with DP):** Items 2 and 4 → total weight = 3 + 5 = 8, Total profit = 2 + 6 = **8**
Greedy (fractional) may yield higher profit, but **not valid for 0/1 Knapsack**.
13. 0/1 Knapsack Problem using Dynamic: Given weights = [2, 3, 4, 5], values = [1, 2, 5, 6], W = 8
Use DP table to fill values. **Answer:** Max profit = **8**

3. Longest Common Subsequence (LCS) problem
Let X = "ABCBDAB", Y = "BDCAB", Using DP:

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

(1) By contradiction, assume zk 6= xm, then by appending xm = yn to Z, we get a common subsequence of X and Y of length k + 1, contradicting the supposed optimality of Z. So zk = xm = yn. Thus, the prefix Zk−1 is a common subsequence of Xm−1 and Yn−1. Next we show that it is an LCS. Suppose for the purpose of contradiction that there exists a common subsequence W of Xm−1 and Yn−1 with length greater than k−1. We can append xm = yn to W and get a common subsequence of X and Y whose length is greater than k, which contradicting the supposed optimality of Z. (2) zk 6= xm implies that Z is a common subsequence of Xm−1 and Y . By contradiction, suppose that there is a common subsequence W of Xm−1 and Y with length greater than k, then W is a common subsequence of Xm

LCS Length: 4, **LCS:** BCAB

4. Branch and Bound for TSP • Used to find the **minimum cost** tour. • Works by exploring all permutations but **prunes** unpromising paths. **Example:** Cities = A, B, C, D, Cost Matrix:

Steps • Start at A • Explore all permutations like A-B-C-D-A • Use cost bound to prune
Optimal Path = A → B → D → C → A **Cost** = 10 + 25 + 30 + 15 = **80**

	A	B	C	D
A	∞	10	15	20
B	10	∞	35	25
C	15	35	∞	30
D	20	25	30	∞

5. KMP (Knuth-Morris-Pratt): Avoids backtracking in the text by using a **prefix table (LPS)**.
Pattern = "ABABC", Text = "ABABABCAB", LPS = [0, 0, 1, 2, 0]
Matching steps • Matches until mismatch, then uses LPS to skip characters • Time complexity: **O(n + m)**
Naïve takes O(nm), KMP is more efficient due to reuse of previous comparisons.
KMP Matching for T = "ABABABCAB", P = "ABABC", LPS for "ABABC": [0, 0, 1, 2, 0]
Matching Process • Match found at index 2 • Time: O(n + m) = O(9 + 5) = O(14)
Improves over Naïve by avoiding rechecks.

6. Assignment Problem: Cost Matrix:

	A	B	C
1	9	2	7
2	6	4	3
3	5	8	1

Use **Hungarian Algorithm**: Optimal Assignment: • Task 1 → B (cost 2)• Task 2 → C (cost 3)• Task 3 → A (cost 5)
Total Cost = 10
Quadratic Assignment Problem Given n facilities and n locations.
Cost depends on • Distance between locations. • Flow between facilities.
Objective: Assign facilities to minimize cost. **Example:** Facilities: F1, F2
Locations: L1, L2 • Cost = Flow × Distance • Try both assignments and pick the one with minimum cost.

7. Las Vegas Algorithm • Always gives correct results • Time may vary due to randomness
Quicksort using Random Pivot • Choose pivot randomly to avoid worst-case ($O(n^2)$)
Expected Time Complexity: Average case: O(n log n)
Compare Las Vegas and Monte Carlo algorithms
Las Vegas • Always correct • Random • Randomized Quick Sort
Monte Carlo • May be incorrect • Fixed • 2-SAT Monte Carlo algorithm
Monte Carlo algorithm: 2-SAT Example: Clauses: ($x1 \vee x2$), ($\neg x1 \vee x3$), ($\neg x2 \vee \neg x3$)
Monte Carlo • Randomly assign truth values • Repeat trials • If satisfied, return; else retry
Probability of success: Increases with number of trials, Each trial: O(n)
Overall: High probability of finding a solution in poly time

9. P, NP, and NP-Complete problems, and prove that the Satisfiability Problem (SAT)
P (Polynomial Time): Class of problems that can be **solved** in polynomial time. **NP (Nondeterministic Polynomial Time):** Class of problems for which a given solution can be **verified** in polynomial time.
NP-Complete: Problems that are: 1. In NP. 2. Every problem in NP can be reduced to it in polynomial time.
SAT Problem: Given a Boolean formula, determine if there is an assignment that satisfies it.
Cook's Theorem (Proof Sketch) • SAT is in NP: given a truth assignment, we can verify in polynomial time.
• Any NP problem can be **reduced to SAT** using a polynomial-time reduction. ∴ **SAT is NP-Complete.**
P, NP, NP-Hard, and NP-Complete + Vertex Cover Vertex Cover Problem: Given graph G(V, E), find the smallest set of vertices covering all edges. • In NP: we can verify a solution in poly time. • Reduce from 3-SAT or CLIQUE problem → NP-Complete.

10. Set Cover Problem: Given • A universe **U** of elements. • A collection of subsets whose union = U.
Goal: Select **the minimum** number of subsets to cover all elements.
Greedy Approximation Algorithm: 1. Pick subset that covers most uncovered elements. 2. Repeat until all elements are covered. **Approximation Ratio:** The greedy algorithm gives an **O(log n)** approximation, where n = size of the universe.

11. Order Notations (Big-O, Ω, and Θ) Big-O (O): Upper bound.
If $f(n) \leq c * g(n)$ for large n, then $f(n) \in O(g(n))$
Big-Omega (Ω): Lower bound. If $f(n) \geq c * g(n)$ for large n, then $f(n) \in \Omega(g(n))$
Big-Theta (Θ): Tight bound. If f(n) is both O(g(n)) and Ω(g(n)), then $f(n) \in \Theta(g(n))$

12. Compare Kruskal's and Prim's Algorithms
Kruskal's This is one of the popular algorithms for finding the minimum spanning tree from a connected, undirected graph. This is a greedy algorithm. The algorithm workflow is as below: • First, it sorts all the edges of the graph by their weights, • Then starts the iterations of finding the spanning tree. • At each iteration, the algorithm adds the next lowest-weight edge one by one, such that the edges picked until now does not form a cycle. • Edge-based • Disjoint Set • O(E log E) • Sparse graphs
Prim's This is also a greedy algorithm. This algorithm has the following workflow: • It starts by selecting an arbitrary vertex and then adding it to the MST. • Then, it repeatedly checks for the minimum edge weight that connects one vertex of MST to another vertex that is not yet in the MST. • This process is continued until all the vertices are included in the MST. • Vertex-based • Priority Queue (Min-Heap) • O(E + V log V) (using Min-Heap) • Dense graphs

14. Solve Weight Capacity = 5, Weights = [2,3,4], Values = [10,20,30] (DP) Using DP:

w\i	**0**	**1**	**2**	**3**
0	0	0	0	0
1	0	0	0	0
2	0	10	10	10
3	0	10	20	20
4	0	10	20	30
5	0	10	30	30

Max Profit = 30