Greedy Algorithm:

A **greedy algorithm** is a problem-solving approach that makes a series of choices, each time selecting the option that seems the best (most "greedy") at the moment, with the hope of finding the overall optimal solution. It works by making local, immediate decisions rather than looking ahead to the bigger picture, assuming that these local choices will lead to a global optimum.

Examples:

1. Coin Change Problem (Greedy Approach):

- **Problem**: Given a set of coin denominations (e.g., ₹1, ₹5, ₹10, ₹20) and a target amount, find the minimum number of coins needed to make the amount.
- **Greedy Approach**: At each step, choose the largest coin denomination that is smaller than or equal to the remaining amount and repeat until the total is reached.
- **Example**: For an amount of ₹49 and coin denominations {₹1, ₹5, ₹10, ₹20}, the greedy solution would be: 20+20+5+1+1+1+1=49.
- **Time Complexity**: O(n), where nnn is the number of coin denominations. If the denominations are sorted, iterating over them takes linear time.
- **Space Complexity**: O(1), as only a few variables are needed to keep track of the current amount and number of coins.

2. Activity Selection Problem:

- **Problem**: Given a set of activities with start and finish times, select the maximum number of non-overlapping activities that can be performed by a single person.
- **Greedy Approach**: At each step, select the activity that finishes the earliest (because this leaves the most time for subsequent activities) and discard overlapping activities.
- **Example**: For activities with start and finish times: {(1, 3), (2, 5), (4, 6), (6, 7), (5, 9)}, the greedy solution would choose activities {(1, 3), (4, 6), (6, 7)} to maximize the number of non-overlapping activities.

- **Time Complexity**: O(nlogn), where n is the number of activities. Sorting the activities by their finish times takes O(nlogn), and then iterating through them to select non-overlapping activities takes O(n).
- **Space Complexity**: O(1), as only a few variables are needed to track the current activity and time.

3. Job Sequencing Problem:

- **Problem**: Given a set of jobs, each with a deadline and a profit if the job is completed before the deadline, schedule the jobs to maximize the total profit. Only one job can be done at a time.
- **Greedy Approach**: Sort the jobs by decreasing profit and schedule each job as late as possible before its deadline, to leave room for other jobs.
- **Example**: For jobs with deadlines and profits: {Job1: (deadline: 2, profit: 100), Job2: (deadline: 1, profit: 50), Job3: (deadline: 2, profit: 20)}, the greedy solution would schedule Job1 at time slot 2 and Job2 at time slot 1 to maximize profit.
- **Time Complexity**: O(nlogn), where n is the number of jobs. Sorting the jobs by profit takes O(nlogn), and scheduling them takes O(n).
- **Space Complexity**: O(n), as we need to store the scheduled jobs and track which time slots are free.

DYNAMIC PROGRAMMING:

Dynamic Programming is a problem-solving technique used to solve optimization problems by breaking them down into smaller overlapping subproblems, solving each subproblem once, and storing the results to avoid redundant computations. DP is typically used in problems where the solution can be constructed efficiently by solving subproblems and combining their solutions (problems with **optimal substructure** and **overlapping subproblems**).

Examples:

1.Coin Change Problem (DP Approach):

- **Problem**: Given a set of coin denominations and a target amount, find the minimum number of coins needed to make the amount.
- **DP Approach**: Use a 1D array where each index i represents the minimum number of coins needed to make amount ii. The solution is built by considering each coin and updating the array for every amount.
- **Example**: For denominations {1, 5, 10} and a target amount of 12, the DP array is filled by calculating the minimum coins needed for every amount up to 12.
- **Time Complexity**: O(n×m), where nnn is the target amount and mmmm is the number of denominations. For each denomination, we update the array for all amounts.
- **Space Complexity**: O(n), since we only need an array of size n+1 to store the minimum number of coins for each amount.

2. 0/1 Knapsack Problem (DP Approach):

- **Problem**: Given a set of items, each with a weight and a value, and a knapsack with a weight capacity, determine the maximum value that can be obtained without exceeding the weight capacity. You can either include an item or exclude it (no fractions allowed).
- **DP Approach**: Use a 2D table where the rows represent the items and the columns represent the weight capacities. Each cell stores the maximum value that can be achieved for a given number of items and weight capacity.
- **Example**: If there are 3 items and the knapsack has a capacity of 50, the table is filled with the optimal values by comparing whether including or excluding each item provides a better result.
- **Time Complexity**: O(nW), where n is the number of items and W is the knapsack's capacity. We compute the optimal value for each weight and item.
- **Space Complexity**: O(nW) for the 2D table. This can be reduced to O(W) by using a single array if only the current and previous row of the table are needed at any point.

3.Bellman-Ford Algorithm (for Shortest Path):

- **Problem**: Given a graph with vertices and weighted edges (including negative weights), find the shortest path from a source vertex to all other vertices.
- **DP Approach**: Use an array where each index represents the shortest distance to a vertex. The algorithm iteratively relaxes each edge, updating the shortest path by checking all edges for each vertex, and it repeats this process for a number of times equal to the number of vertices minus one.
- **Example**: For a graph with edges {(A, B, 1), (B, C, 2), (A, C, 4), (C, D, -1)}, Bellman-Ford will compute the shortest paths from vertex A by iterating over the edges multiple times.
- **Time Complexity**: O(V×E), where V is the number of vertices and E is the number of edges. Each edge is checked for every vertex in the graph.
- **Space Complexity**: O(V), for storing the array of shortest distances.