1. Greedy Algorithms

Greedy algorithms build a solution piece by piece, choosing the next piece that offers the most immediate benefit or lowest cost, without considering future consequences. Greedy algorithms are often used when a problem has an optimal substructure (i.e., a globally optimal solution can be achieved by making locally optimal choices).

Example: Suppose you want to make change for a certain amount of money using the fewest coins. Using a greedy approach, you would start by selecting the largest coin that does not exceed the remaining amount and continue this process until you reach zero.

For example, for an amount of 63 using coin denominations of 25, 10, 5, and 1:

1. Take one 25 coin (remaining amount: 38).

2. Take another 25 coin (remaining amount: 13).

3. Take one 10 coin (remaining amount: 3).

4. Take three 1 coins (remaining amount: 0).

The greedy approach results in the fewest coins: 2x25, 1x10, and 3x1.

Complexity:

Time Complexity: Depends on the specific problem, but typically O(n log n) to O(n).

Space Complexity: Often O(1) or O(n), as many greedy algorithms work in-place or use only a few variables.

Data Structures Used: Often lists or arrays; some problems may require priority queues or heaps.

---

2. Dynamic Programming (DP)

Dynamic programming (DP) is an optimization technique that solves complex problems by breaking them down into simpler subproblems and storing the solutions of subproblems to avoid redundant calculations.

Types of DP:

1. Memoization (Top-Down): Recursively solve each subproblem and store results in a table to prevent recalculations.

2. Tabulation (Bottom-Up): Iteratively build solutions from the smallest subproblem to the largest, storing results in a table.

Common Algorithms:

Fibonacci Sequence (finding nth Fibonacci number)

Knapsack Problem (finding maximum value in a set of items with weight limits)

Longest Common Subsequence (LCS) (finding longest matching subsequence between two sequences)

Shortest Path Problems (finding shortest path in weighted graphs, e.g., Bellman-Ford, Floyd-Warshall)

Matrix Chain Multiplication (minimizing multiplication cost in a chain of matrices)

---

3. Complexity and Data Structures for DP Algorithms

Time Complexity: Typically, DP algorithms are optimized to O(n^2) or better, though this varies based on the problem. Space Complexity: Often O(n) or O(n^2), depending on whether we store intermediate results in a 1D or 2D array. Data Structures Used: Arrays, matrices, and hash tables (for memoization).

Here's a breakdown of common dynamic programming (DP) algorithms, along with their time complexity, space complexity, and the type of DP (memoization or tabulation) they typically use:

---

4))Fibonacci Sequence

Problem: Calculate the nth Fibonacci number.

Memoization (Top-Down): Recursive approach that stores the result of each Fibonacci number calculation to avoid redundant calculations.

Tabulation (Bottom-Up): Iterative approach that calculates each Fibonacci number up to n, storing them in a table or array.

Complexities:

Time Complexity: O(n)

Space Complexity: O(n) for memoization; O(1) for tabulation if optimized.

Data Structures Used: Array (or variables if space-optimized).

---

1. 0/1 Knapsack Problem

Problem: Given items with weights and values, maximize the value in a knapsack of limited capacity.

Tabulation (Bottom-Up): Fill a DP table where each entry represents the maximum value for a given capacity up to a specific item.

Memoization (Top-Down): Recursive approach with memoization to store results of subproblems (capacity and items considered).

Complexities:

Time Complexity: O(n * W), where n is the number of items and W is the knapsack capacity.

Space Complexity: O(n * W) for standard DP table; O(W) if using a single-row optimized approach.

Data Structures Used: 2D array for storing results of subproblems.

2. Longest Common Subsequence (LCS)

Problem: Find the longest subsequence common to two sequences.

Tabulation (Bottom-Up): Fill a DP table where each entry represents the LCS length for a specific pair of indices in the two sequences.

Memoization (Top-Down): Recursive approach storing results of LCS lengths for specific indices in both sequences.

Complexities:

Time Complexity: O(m * n), where m and n are the lengths of the two sequences.

Space Complexity: O(m * n) for a 2D DP table; O(min(m, n)) if optimized.

Data Structures Used: 2D array for standard approach; 1D array if space-optimized.

---

3. Shortest Path in Weighted Graph (Bellman-Ford)

Problem: Find the shortest path from a source vertex to all other vertices in a graph with edge weights.

Tabulation (Bottom-Up): Iteratively relax all edges, ensuring each vertex has the minimum distance value by the end of iterations.

Complexities:

Time Complexity: O(V * E), where V is the number of vertices and E is the number of edges.

Space Complexity: O(V) for distance storage.

Data Structures Used: Array for distances.

---

4. Matrix Chain Multiplication

Problem: Determine the most efficient way to multiply a chain of matrices by minimizing the multiplication cost.

Tabulation (Bottom-Up): Use a 2D DP table where each entry represents the minimum cost of multiplying matrices between certain indices.

Complexities:

Time Complexity: O(n^3), where n is the number of matrices.

Space Complexity: O(n^2) for a 2D DP table.

Data Structures Used: 2D array.

---

5. Edit Distance (Levenshtein Distance)

Problem: Find the minimum number of operations (insertions, deletions, substitutions) needed to transform one string into another.

Tabulation (Bottom-Up): Use a 2D DP table where each entry represents the minimum number of edits required for specific substrings.

Memoization (Top-Down): Recursive approach with memoization to store results for specific pairs of substrings.

Complexities:

Time Complexity: O(m * n), where m and n are the lengths of the two strings.

Space Complexity: O(m * n) for a 2D DP table; O(min(m, n)) if optimized.

Data Structures Used: 2D array or 1D array if space-optimized.