DP Concepts Revision Guide

i want revice all the concept dp

To effectively revise all the concepts of **Dynamic Programming (DP)**, here's a structured approach:

---

1. Key Concepts of DP

- **Overlapping Subproblems**: Solve the same subproblem multiple times.
- **Optimal Substructure**: The solution to a problem can be constructed from solutions to its subproblems.
- **Memoization vs Tabulation**:
    - **Memoization**: Top-down approach (recursive with caching).
    - **Tabulation**: Bottom-up approach (iterative with table).

---

2. Core Topics to Cover

Basic Problems (Foundation):

- Fibonacci Sequence
- Climbing Stairs
- Min Cost Path in a Grid
- Subset Sum Problem
- 0/1 Knapsack Problem

Intermediate Problems:

- Longest Common Subsequence (LCS)
- Longest Increasing Subsequence (LIS)
- Matrix Chain Multiplication
- Edit Distance
- Partition Equal Subset Sum

Advanced Problems:

- Coin Change (Minimum and Number of Ways)
- Rod Cutting
- Word Break Problem
- Palindromic Subsequence
- Maximum Sum Increasing Subsequence
- Maximum Profit in Job Scheduling

2D DP Problems:

- Unique Paths in a Grid
- Maximum Square Sub-matrix of 1s
- Cherry Pickup

---

3. Types of DP Approaches

- **1D DP**: Problems where you use a 1D array for storage (e.g., Fibonacci).
- **2D DP**: Problems involving a grid or two sequences (e.g., LCS, Edit Distance).
- **Bitmask DP**: Used in problems involving subsets and states (e.g., Traveling Salesman Problem).
- **DP on Trees**: Solve problems related to trees (e.g., Diameter of a Tree, Maximum Path Sum).
- **DP with States**: Problems involving multiple states (e.g., Egg Dropping Problem).

---

4. Steps to Solve a DP Problem

1. **Understand the Problem**: Break it into subproblems.
2. **Define the State**: Identify what `` `dp[i]` `` or `` `dp[i][j]` `` represents.
3. **State Transition**: Derive the relation between the states.
4. **Base Cases**: Define the starting point.
5. **Implementation**: Choose memoization or tabulation.

---

5. Revision Plan

Week 1: Basics and 1D DP

- Solve easy problems like Fibonacci, Climbing Stairs, etc.

Week 2: 2D DP and Intermediate Problems

- Focus on LCS, LIS, and matrix-based problems.

Week 3: Advanced Problems and Special Techniques

- Practice problems like Word Break, Bitmask DP, and DP on Trees.

Week 4: Contest Practice

- Solve mixed problems from platforms like LeetCode, Codeforces, and AtCoder.

---

6. Resources for Practice

- **Books**:
  - *Introduction to Algorithms* by Cormen (CLRS)
  - *Dynamic Programming for Coding Interviews* by Aditya Bhargava
- **Online Platforms**:
  - LeetCode (DP Category)
  - Codeforces (Div2 D/E problems)
  - AtCoder DP Contest
  - GeeksforGeeks

Would you like a detailed note or any specific topic explanation? Let me know!

give all related algorithm and methods

Here's a comprehensive guide to all the **Dynamic Programming (DP) algorithms and methods**, broken down by problem type, approach, and associated algorithms.

---

1. Basic DP Algorithms

1.1 Fibonacci Numbers

- **State**: `dp[i]` = i-th Fibonacci number
- **Recurrence Relation**: `dp[i] = dp[i-1] + dp[i-2]`
- **Time Complexity**: $O(n)$

1.2 Climbing Stairs

- **State**: `dp[i]` = number of ways to reach step `i`
- **Recurrence Relation**: `dp[i] = dp[i-1] + dp[i-2]`

---

2. 0/1 Knapsack and Variants

2.1 0/1 Knapsack

- **State**: `dp[i][w]` = max value attainable with `i` items and capacity `w`
- **Recurrence Relation**:
  - If item can be included: `dp[i][w] = max(dp[i-1][w], dp[i-1][w-weight[i]] + value[i])`
  - Otherwise: `dp[i][w] = dp[i-1][w]`

2.2 Unbounded Knapsack

- **State**: `dp[i][w]` = max value using items with unlimited quantity
- **Recurrence Relation**: `dp[i][w] = max(dp[i-1][w], dp[i][w-weight[i]] + value[i])`

2.3 Subset Sum

- **State**: `dp[i][j]` = whether sum `j` can be formed using first `i` elements
- **Recurrence Relation**:
  - If current element is included: `dp[i][j] = dp[i-1][j-arr[i]]`
  - Otherwise: `dp[i][j] = dp[i-1][j]`

---

3. Sequence Alignment and String Problems

3.1 Longest Common Subsequence (LCS)

- **State**: `dp[i][j]` = LCS length of first `i` characters of `X` and first `j` of `Y`
- **Recurrence Relation**:
    - If `X[i-1] == Y[j-1]`: `dp[i][j] = 1 + dp[i-1][j-1]`
    - Else: `dp[i][j] = max(dp[i-1][j], dp[i][j-1])`

3.2 Longest Increasing Subsequence (LIS)

- **State**: `dp[i]` = length of LIS ending at index `i`
- **Recurrence Relation**: `dp[i] = max(dp[j] + 1)` for all `j < i` and `arr[j] < arr[i]`

3.3 Edit Distance

- **State**: `dp[i][j]` = minimum operations to convert first `i` chars of `X` to first `j` of `Y`
- **Recurrence Relation**:
    - If `X[i-1] == Y[j-1]`: `dp[i][j] = dp[i-1][j-1]`
    - Else: `dp[i][j] = 1 + min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1])`

---

4. Matrix and Grid-Based Problems

4.1 Min Cost Path in a Grid

- **State**: `dp[i][j]` = minimum cost to reach cell `(i, j)`
- **Recurrence Relation**: `dp[i][j] = grid[i][j] + min(dp[i-1][j], dp[i][j-1])`

4.2 Unique Paths

- **State**: `dp[i][j]` = number of unique paths to cell `(i, j)`
- **Recurrence Relation**: `dp[i][j] = dp[i-1][j] + dp[i][j-1]`

---

5. Partition Problems

5.1 Partition Equal Subset Sum

- **State**: `dp[i][j]` = whether subset with sum `j` can be formed with first `i` elements
- **Recurrence Relation**: Similar to Subset Sum problem.

5.2 Palindromic Subsequence

- **State**: `dp[i][j]` = length of longest palindromic subsequence in substring `s[i:j]`
- **Recurrence Relation**:
    - If `s[i] == s[j]`: `dp[i][j] = dp[i+1][j-1] + 2`
    - Else: `dp[i][j] = max(dp[i+1][j], dp[i][j-1])`

---

6. Advanced DP Techniques

6.1 Matrix Chain Multiplication

- **State**: `dp[i][j]` = minimum cost to multiply matrices from `i` to `j`
- **Recurrence Relation**: `dp[i][j] = min(dp[i][k] + dp[k+1][j] + cost of multiplying i, k, j)`

6.2 Egg Dropping Problem

- **State**: `dp[e][f]` = minimum number of attempts with `e` eggs and `f` floors
- **Recurrence Relation**:
    - `dp[e][f] = 1 + min(max(dp[e-1][x-1], dp[e][f-x]))` for all `1 <= x <= f`

6.3 Bitmask DP (e.g., Traveling Salesman Problem)

- **State**: `dp[mask][i]` = minimum cost to visit all nodes in `mask` ending at `i`
- **Recurrence Relation**: `dp[mask][i] = min(dp[prev_mask][j] + cost[j][i])`

---

7. Tree DP

7.1 Maximum Path Sum in a Tree

- **State**: `dp[node]` = maximum path sum rooted at `node`
- **Recurrence Relation**: Use DFS to calculate sums from children.

7.2 Diameter of a Tree

- **State**: Track the maximum depth and diameter during DFS traversal.

---

8. Resource Allocation Problems

8.1 Rod Cutting

- **State**: `dp[i]` = maximum revenue from a rod of length `i`
- **Recurrence Relation**: `dp[i] = max(price[j] + dp[i-j-1])` for all `j < i`

8.2 Coin Change

- **Number of Ways**:
    - **State**: `dp[i]` = number of ways to make sum `i`
    - **Recurrence**: `dp[i] = dp[i] + dp[i-coin]`
- **Minimum Coins**:
    - **State**: `dp[i]` = minimum coins to make sum `i`
    - **Recurrence**: `dp[i] = min(dp[i], dp[i-coin] + 1)`

General Methods for Problem-Solving:

1. **Identify the Subproblem**: Define `dp[i]` or `dp[i][j]` clearly.
2. **Formulate Recurrence Relation**.
3. **Decide Base Cases**.
4. **Choose Optimization (Tabulation vs Memoization)**.

Let me know if you need any specific problem or concept in detail!