# Dynamic programming

## Introduction

Dynamic Programming (DP) is usually used to solve optimization problems. The only way to get better at DP is to practice. It takes some amount of practice to be able to recognize that a problem can be solved by DP.

### When to Use DP?

Use DP when your problem has:

1. **Overlapping Subproblems** – same subproblems are solved multiple times.

2. **Optimal Substructure** – the solution to the overall problem depends on the solution to subproblems.

✨ A classic sign is when you use recursion and get TLE (Time Limit Exceeded) because you're solving the same thing again and again.

### Two Main Approaches:

#### 1. Top-Down (Memoization)

- Use **recursion**.
- Store answers of already solved subproblems in a table (usually an array or map).

**Example: Fibonacci (Top-Down)**

```
int fib(int n, vector<int>& dp) {
    if (n <= 1) return n;
```

```
    if (dp[n] != -1) return dp[n];  // Already solved
    return dp[n] = fib(n-1, dp) + fib(n-2, dp);
}
```

**2. Bottom-Up (Tabulation)**

- Solve the smallest subproblems first.
- Build the solution iteratively.

Example: **Fibonacci (Bottom-Up)**

```
int fib(int n) {
    vector<int> dp(n+1);
    dp[0] = 0, dp[1] = 1;
    for(int i=2; i<=n; i++)
        dp[i] = dp[i-1] + dp[i-2];
    return dp[n];
}
```

# Classic Problems in DP

Here are some famous ones you should try:

| Problem | Description |
|---|---|
| Fibonacci | Return n-th Fibonacci number |
| 0/1 Knapsack | Max value with weight constraints |
| Coin Change | Minimum coins to make amount |

| | |
|---|---|
| Longest Common Subsequence | Length of common subsequence in two strings |
| Longest Increasing Subsequence | Length of increasing subsequence |
| Matrix Chain Multiplication | Min operations to multiply matrices |
| Edit Distance | Min changes to convert one string to another |

## Key Concepts You'll Often Use in DP

- Recursion + Memoization
- Iterative table building
- State definition (dp[i][j] meaning)
- Space Optimization (use only last few states)

## Tips to Master DP

1. **Understand the problem** thoroughly.
2. **Find the recurrence relation**.
3. Decide: Memoization or Tabulation?
4. **Draw the recursion tree** (very helpful).
5. **Practice, practice, practice**!

# Graph

## Introduction

A graph is a structure containing a set of objects (nodes or vertices) where there can be edges between these nodes/vertices. Edges can be directed or undirected and can optionally have values (a weighted graph). Trees are undirected graphs in which any two vertices are connected by exactly one edge and there can be no cycles in the graph.

Graphs are commonly used to model relationship between unordered entities, such as

- Friendship between people - Each node is a person and edges between nodes represent that these two people are friends.
- Distances between locations - Each node is a location and the edge between nodes represent that these locations are connected. The value of the edge represents the distance.

Be familiar with the various graph representations, graph search algorithms and their time and space complexities.

## Types of Graphs

| Type | Description |
| --- | --- |
| **Undirected Graph** | Edges have no direction (a road goes both ways) |

| | |
|---|---|
| **Directed Graph (Digraph)** | Edges have direction (like one-way roads) |
| **Weighted Graph** | Edges have weights/costs (like distance or time) |
| **Unweighted Graph** | All edges have the same weight (or no weight) |
| **Cyclic / Acyclic** | A cycle means a path that starts and ends at the same node |
| **Connected / Disconnected** | A connected graph means all nodes are reachable from each other |
| **Tree** | A connected, acyclic graph with **n nodes and n-1 edges** |

# Graph Representation

### 1. Adjacency Matrix

```
int adj[n][n]; // 0 or 1 if edge exists
```

### 2. Adjacency List (Most common)

```
vector<int> adj[n]; // adj[i] = list of neighbors of node i
```

### 3. Edge List

```
vector<pair<int, int>> edges; // list of all edges
```

# Graph Traversal

**1. Breadth First Search (BFS) – Like level order traversal in trees**

- Uses **queue**
- Best for shortest path in unweighted graphs

**2. Depth First Search (DFS) – Like preorder traversal**

- Uses **stack** or **recursion**
- Good for checking cycles, connected components

# Important Algorithms in Graphs

| Algorithm | Purpose |
|---|---|
| **BFS / DFS** | Traversal |
| **Dijkstra's** | Shortest path in **weighted graph** with positive weights |
| **Bellman-Ford** | Shortest path, handles negative weights |
| **Floyd-Warshall** | All-pairs shortest path |
| **Topological Sort** | Order of tasks (DAG only) |
| **Kruskal's / Prim's** | Minimum Spanning Tree (MST) |
| **Union-Find** | Detect cycle, used in Kruskal's |
| **Tarjan's / Kosaraju's** | Strongly connected components (SCC) |

# Applications of Graphs

- Maps & GPS (shortest path)
- Social Networks (friend suggestions)
- Computer Networks (routing)
- Task Scheduling (using topological sort)
- Game maps (BFS for pathfinding)
- Web crawling (DFS)

# Key Graph Interview Problems

1. BFS & DFS on matrix
2. Detect cycle in graph (DFS + visited)
3. Number of islands (BFS/DFS on grid)
4. Shortest Path in Binary Matrix (BFS)
5. Topological Sort (Kahn's Algo)
6. Dijkstra's Algorithm
7. Minimum Spanning Tree

~ Mitanshi Jain