DAA   unit-3   Soham Phatak   1

Dynamic Programming : method for solving complex problems by breaking them into smaller overlapping sub problems.

→ Problem is divided into sub problems which are solved simultaneously and solutions are stored to avoid redundant computations.

E.g.

```
fib(int n) {
    vector<int> dp(n+1,0);          → sub.problems stored
    dp[1] = 1;
    for (i=2 to n)
        dp[i] = dp[i-1] + dp[i-2];  } → solved one by one.
    return dp[n];                   → main problem
}
```

- if general solⁿ would've been considered (Recursive solⁿ)
  T.C → $O(2^n)$
  Here with DP approach T.C → $O(n)$

General Strategy:

1) Define Subproblems
2) Formulate Recurrence Solⁿ
3) Memoization / Tabulation to store results ← define Base Case before this
4) Solve Org. solution

D.P.	Greedy
1) Breaks problems into sub-problems and stores their solutions.	1) Makes locally optimal choices to build solution
2) Guarantees globally optimal soln.	2) May or may not.
3) Effecient but time depends on subproblems	3) Generally faster.
4) Problems having overloping subproblems.	4) Problems with straightforwd decisions
5) Medium space complexity	5) Low Space Complexity

(*) Principle of Optimality

→ If an optimal solution to a problem contains subproblems, the sol^n to these subproblems must also be optimal.

(·) Shortest path to $A \rightarrow C$ if passes through B then $A \rightarrow B$ should also be the shortest path to B

→ This is the fondation of DP and allows solving prob. by breaking them int subproblems.

(*) DP approach is optimization technique

1) Efficient Computation : avoids redundant calculations by storing subproblem sol^n.

2) Global optimal sol^n : ensures globally optimal sol^n by considering all possible sol^n.

3) Resource optimization : saves both effort to calculate and time (tabulization/memoi..) by systematic solving sub prob.

4) Wide Applications : shortest path, knapsack, matrix chain multiplication, etc.

(*) Limitations

(·) 1) High Space Complexity : Due to sol^n storing

(·) 2) overhead of state storage : requires complex state (visited/not visited) etc. representations for large problems

(·) 3) Difficulty to identify subproblems in case problem has no overlapping / optimal structure.

(·) 4) Difficult to define recurr. relations & maintaining Tables.

0/1 Knapsack Problem

➢ Problem Statement:

- You are given a set of items: 1. A **weight** w[i]
  2. A **value** v[i]

- You need to determine the **maximum total value** you can carry in a knapsack with a maximum weight **capacity w**. Each item **(total n items)** can either be **included (1) or excluded (0)**, hence the name 0/1 Knapsack.

➢ Dynamic Programming Approach: (video)

- A table **dp[n+1][w+1]** → **dp[i][j]** represents the maximum value obtainable with the first **i** items and a knapsack capacity **j**

- The value of **dp[i][j]** is decided by:
  1. Include the item (if the weight allows).
  2. Exclude the item.
  3. Take the maximum of these two choices.

➢ Algorithm:

1. **Initialize** the dp table with all elements as 0.
2. For each item i, **iterate** through all capacities j from 0 to w:
   - If $w[i] \leq j$:

$$dp[i][j] = \max(dp[i-1][j], v[i] + dp[i-1][j-w[i]])$$

   - Otherwise:

$$dp[i][j] = dp[i-1][j]$$

3. **Return** dp[n][w], the value at the bottom-right corner of the table.

➢ Time Complexity and Comparison:

DP Approach:

$$O(n \times W)$$

Where n is the number of items and W is the knapsack capacity.

Normal Approach:

$$O(2^n)$$

Try all possible subsets of items, checking if the total weight is within W. This has an exponential complexity

Aspect	DP Approach	Brute Force
Time Complexity	$O(n \times W)$	$O(2^n)$
Space Complexity	$O(n \times W)$	$O(n)$
Efficiency	Highly efficient	Not practical

➢ Applications:

- **Resource Allocation:** Allocating limited resources (e.g., budget, manpower) to maximize output.
- **Investment Portfolio:** Choosing the best set of stocks to invest in, given a budget and expected returns.
- **Cargo Optimization:** Selecting the most valuable items that fit within the weight limit of a vehicle.
- **Cutting-Edge AI:** In decision-making systems where constraints like time/budget need to be respected.
- **Time Management:** Selecting tasks to perform within a limited time while maximizing productivity.

Coin Change Problem

Problem Statement:

You are given a set of coins with denominations $\{c_1, c_2, \ldots, c_n\}$ and you need to make a total amount T.

1. **Total Number of Ways**: Find how many ways you can make the total amount T using the coins (unlimited supply of each coin).
2. **Minimum Coins Needed**: Find the minimum number of coins required to make the total amount T.

Dynamic Programming Approach: (video1) (video2)

Define a DP table **dp[i][j]**, where:

- i represents the first i coins considered. (i → 0 to n)
- j represents the target amount. (j → 0 to T)

Each cell **dp[i][j]** will:

1. Represent the **number of ways** to make up amount jjj using the first i coins in **case 1**.
2. Represent the **minimum coins** needed to make amount jjj using the first iii coins in **case 2**.

Base Cases:

1. $dp[i][0] = 1$ for all i: There is **1 way** to make amount 0 (use no coins).
2. $dp[0][j] = \infty$ for the minimum coin problem (impossible to make any amount without coins).
3. $dp[0][j] = 0$ for the total ways problem (no ways to make a positive amount without coins).

Algorithm:

- **Initialize** the dp table with all elements as 0.
- **coins[n]** is the array of $\{c_1, c_2, \ldots, c_n\}$
- For each item i, **iterate** through all capacities j from 0 to w:
  - If the current coin $\text{coins}[i-1]$ can be used ($j \geq \text{coins}[i-1]$):
    - For Total Ways:

$$dp[i][j] = dp[i-1][j] + dp[i][j - \text{coins}[i-1]]$$

      (Exclude the coin + Include the coin).

    - For Minimum Coins:

$\mapsto i$

$$dp[i][j] = \min(dp[i-1][j], 1 + dp[i][j - \text{coins}[i-1]])$$

      (Exclude the coin OR include the coin).

  - Otherwise ($j < \text{coins}[i-1]$):

$$dp[i][j] = dp[i-1][j]$$

- **Return** dp[n][T], the value at the bottom-right corner of the table.

Time Complexity and Comparison: same as Knapsack Problem, W becomes T

Applications:

- **Cashier Systems**: Determining the minimum coins or bills needed for a specific amount of change.
- **Making Combinations**: Calculating the total number of ways to make combinations for recipes, game scores, etc.
- **Budget Allocation**: Allocating resources (money, points) efficiently in projects or activities.
- **Optimization Problems**: Used in network flow or logistics to distribute resources efficiently.
- **Cryptography**: Used in certain encryption algorithms involving denominations.

Bellman-Ford Algorithm

Problem Statement:

You are given a graph represented as G=(V, E), where V is the set of vertices, and E is the set of weighted edges (u, v, w) with w as the weight of the edge between u and v.

The task is to find the shortest path from a **source vertex** S to all other vertices in the graph. The graph may **contain negative weights**, but **not negative weight cycles**.

Dynamic Programming Approach: (video)

Use a **distance table** dist[V] where dist[i] stores the shortest distance from the source S to vertex i. The Bellman-Ford algorithm is based on **edge relaxation**:

1. For each edge (u, v, w), update ***dist[v] = min(dist[v], dist[u]+w)***.
2. Repeat this process **V−1 times** (number of vertices - 1).
3. Detect negative weight cycles by running a final iteration. If any edge can still be relaxed, a negative weight cycle exists.

Algorithm:

1. **Initialize** the distance table:
   - dist[S] = 0 (distance to the source is 0).
   - dist[i]=∞ for all other vertices i.
2. Perform |V|−1 iterations:
   - For each edge (u, v, w), **relax the edge**:

$$dist[v]=min(dist[v], dist[u]+w)$$

3. Check for **negative weight cycles** (after V-1 iterations):
   - For each edge (u, v, w), if $dist[v] > dist[u] + w$, there is a negative weight cycle.

Time Complexity and Comparison:

DP Approach:

$$O(V\times E)$$

- V: Number of vertices.
- E: Number of edges.

Aspect	Bellman-Ford	Dijkstra
Negative Weights	Supported	Not supported
Time Complexity	$O(V \times E)$	$O((V + E) \log V)$
Efficiency	Slower	Faster for non-negative weights

Applications:

- **Routing Protocols**: Used in networking protocols like RIP (Routing Information Protocol) to calculate shortest paths in networks.
- **Transportation Planning**: Optimizing routes in systems with mixed positive and negative costs.
- **Finance**: Identifying arbitrage opportunities in currency trading by detecting negative weight cycles.
- **Project Management**: Used in PERT (Program Evaluation Review Technique) to calculate shortest time to complete a project with dependencies.
- **AI and Robotics**: Pathfinding in weighted graphs, especially in dynamic or uncertain environments.

Soham Phatak

Multistage Graph Problem (Forward Computation)

➢ Problem Statement:

A **multistage graph** is a directed graph in which the vertices are divided into multiple stages.

Every edge connects a vertex in one stage to a vertex in the next stage.

The goal is to find the shortest path from a **source vertex** S in the first stage to a **destination vertex** D in the last stage.

➢ Dynamic Programming Approach: (video – backward computation)

We calculate the shortest path from S to D by computing the minimum cost at each stage in a **forward manner**.

1. Divide the graph into K stages.
2. Use a DP aray **cost[v]** to store the minimum **cost to reach vertex v** from the source S.
3. Use an array **source[v]**, where source[i] will denote which vertex from the previous stage which should be the source to vertex i for optimal cost.
4. Iterate through each stage starting from the first stage, and compute cost[v] for each vertex v in that stage.

➢ Algorithm:

1. Initialize **cost[S]=0** and **cost[v]=∞** for all other vertices v.
2. **For each** vertex u in the current stage:
   - For every outgoing edge (u, v, w):
     - Update ***cost[v]=min(cost[v], cost[u]+w)***
     - If **value gets** updated to cost[u]+w, store ***source[v] = u***
3. **Repeat** until the last stage is processed.
4. **Return** cost[D].
5. **Backtrack source[D]** till we get the optimal path from S to D.

```
vector<int> path; // To store the traced path
int i = D;        // Start from the destination node
path.push_back(D);

// Backtrack the path using the source array
while (i != S) {
    i = source[i];
    path.push_back(i);
}

// Reverse the path to get it from source to destination
reverse(path.begin(), path.end());
```

➢ Time Complexity and Comparison:

Aspect	DP Approach	Brute Force
Time Complexity	$O(E)$	$O(V!)$
Space Complexity	$O(V)$	$O(V)$
Efficiency	Polynomial, efficient	Exponential, impractical

➢ Applications:

- **Telecommunication:** Finding the shortest transmission path through layered networks.
- **Project Scheduling:** Optimizing dependencies and resource allocations with sequential tasks.
- **Game Development:** Pathfinding in games with level-based progressions.
- **Manufacturing Processes:** Minimizing costs in sequential stages of production.
- **Transportation Networks:** Optimizing paths in multistage transportation systems such as railways, flights, or delivery networks.

▪ **Backward Computation** or Backward Dynamic Programming is the approach where we start from the last vertex and work backward towards the first vertex.

▪ You solve the problem by considering the final goal (destination) first and compute the optimal solutions for preceding stages, working backward to the starting point.

Traveling Salesperson Problem

➢ Problem Statement:

Given **N cities** and the distance between every pair of cities, the goal of the Traveling Salesperson Problem (TSP) is to find the shortest possible route that:

- Visits every city exactly once.
- Returns to the starting city.

➢ Dynamic Programming Approach and Algorithm: (video) (video)

- **n**: Number of cities.
- **c[i][j]**: Cost of traveling from city i to city j.

1. *State Representation:*
   Use a **DP table dp[i][S]** where:
   - i is the current city.
   - S represents the subset of cities to visit, for example {2,3,4} = 0111, {1,4} = 1001, ∅ = 0000
   - **Starting city is not part of S.**
2. *Initialization:*
   - **dp[i][∅] = c[i][0]** for all i.
3. *Recurrence Relation:*
   - For each subset S, and for each **city i not in S**:

$$\boxed{dp[i][S] = \min_{k \in S}\{c[i][k] + dp[k][S \setminus \{k\}]\}}$$

4. *Iterative Computation:*
   - Start with smaller subsets S and calculate dp[i][S] for all cities i and subsets S.
5. *Final Solution:*
   - The final answer is: ***dp[starting city][S]***
   - Here, S is the set of all the cities to visit.

➢ Time Complexity and Comparison:

Aspect	DP Approach	Brute Force
Time Complexity	$O(N^2 \times 2^N)$	$O(N!)$
Space Complexity	$O(N \times 2^N)$	$O(1)$
Efficiency	More efficient	Computationally expensive

➢ Applications:

- **Logistics and Delivery:** Optimizing routes for delivery trucks to minimize travel time and fuel costs.
- **Manufacturing:** Sequencing operations on machines in a factory to minimize setup costs.
- **Robotics:** Path planning for automated robots covering multiple locations.
- **Circuit Design:** Minimizing the length of wiring between components in VLSI circuits.
- **Travel Planning:** Organizing efficient tours for travel agents and tourists.