Definition of Dynamic Programming (DP):

Dynamic Programming (DP) is a **mathematical optimization technique** and a programming method used to solve problems by **breaking them into smaller, overlapping subproblems** and solving each subproblem only once, storing its results for future use.

Key Characteristics:

1. **Overlapping Subproblems**:
   Problems can be divided into smaller subproblems that are solved multiple times.
2. **Optimal Substructure**:
   The optimal solution of a problem can be constructed from the optimal solutions of its subproblems.

Steps to Apply DP:

1. **Identify the Subproblems**: Define smaller problems that make up the larger problem.
2. **Formulate a Recurrence Relation**: Define how the solution of the current problem depends on its subproblems.
3. **Base Cases**: Specify the solution for the simplest version of the problem.
4. **Solve Using Tabulation or Memoization**:
   - **Tabulation (Bottom-Up)**: Build the solution iteratively.
   - **Memoization (Top-Down)**: Solve recursively and cache results.

The Knapsack problem is an optimization problem where you are given a set of items, each with a weight and a value. The goal is to maximize the total value of items placed in a knapsack without exceeding its weight capacity.

Problem Statement:

You are given:

- A set of n items, each with a weight w[i] and a value v[i].
- A maximum capacity W of the knapsack.

The objective is to maximize the total value of items selected without exceeding the weight capacity W.

Dynamic Programming Approach:

1. Define the DP array: Let dp[i][j] represent the maximum value that can be achieved using the first i items with a knapsack capacity of j.

2. Recurrence relations: Include item i: If w[i] <= j, we can include the item. The formula is: dp[i][j] = v[i] + dp[i-1][j-w[i]]

   Exclude item i: Otherwise, we exclude the item. The formula is: dp[i][j] = dp[i-1][j]

   Combine both cases: dp[i][j] = max(dp[i-1][j], v[i] + dp[i-1][j-w[i]]) if w[i] <= j

3. Base case: If no items are available (i = 0) or the knapsack capacity is 0 (j = 0), then: dp[i][j] = 0

4. Final answer: The solution to the problem is stored in dp[n][W], where n is the number of items and W is the knapsack's capacity.

Code Implementation

```
def knapsack(values, weights, capacity):
    n = len(values)
    dp = [[0] * (capacity + 1) for _ in range(n + 1)]

    for i in range(1, n + 1):
        for j in range(1, capacity + 1):
            if weights[i - 1] <= j:
                dp[i][j] = max(dp[i - 1][j], values[i - 1] + dp[i - 1][j -
weights[i - 1]])
            else:
                dp[i][j] = dp[i - 1][j]

    return dp[n][capacity]
```

Optimized Space Complexity:

Instead of using a 2D DP array, we can use a 1D array because the values of dp[i][j] depend only on dp[i-1][j].

Optimized Code:

```
def knapsack(values, weights, capacity):
    n = len(values)
    dp = [0] * (capacity + 1)

    for i in range(n):
        for j in range(capacity, weights[i] - 1, -1):
            dp[j] = max(dp[j], values[i] + dp[j - weights[i]])

    return dp[capacity]
```

Complexity Analysis:

1. Time complexity: O(n * W), where n is the number of items and W is the knapsack capacity.
2. Space complexity:
    - Using a 2D DP array: O(n * W)
    - Using a 1D DP array: O(W)

Floyd-Warshall Algorithm:

Purpose:

The Floyd-Warshall algorithm finds the shortest paths between all pairs of nodes in a graph. It works for graphs with both positive and negative edge weights but does not work if there is a negative weight cycle.

Algorithm Steps:

1. **Initialization**:
    - Use an adjacency matrix `dist`, where `dist[i][j]` represents the weight of the edge between node `i` and node `j`. If there is no direct edge, set `dist[i][j]` to infinity (`inf`).
    - Set `dist[i][i] = 0` for all nodes.
2. **Dynamic Programming Approach**:
    - Iterate over each intermediate node kk.

For every pair of nodes ii and jj, update `dist[i][j]` as:
dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

-
- This checks if including kk as an intermediate node results in a shorter path.

3. **Negative Weight Cycle Check**:
    - After the algorithm, if `dist[i][i] < 0` for any ii, the graph contains a negative weight cycle.

```python
def floyd_warshall(graph):

    n = len(graph)

    dist = [[float('inf')] * n for _ in range(n)]


    for i in range(n):

        for j in range(n):

            dist[i][j] = graph[i][j]


    for k in range(n):

        for i in range(n):

            for j in range(n):

                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])


    return dist
```

Complexity:

- **Time Complexity**: O(V^3)
- **Space Complexity**: O(V^2)

Bellman-Ford Algorithm

Purpose:
The Bellman-Ford algorithm finds the shortest path from a single source node to all other nodes in a graph. It works with graphs containing negative edge weights and can also detect negative weight cycles.

Algorithm Steps:

1. Initialization:
   Create a dist array where dist[src] = 0 and dist[v] = infinity for all other vertices v.

2. Relax Edges:
   Repeat V-1 times (where V is the number of vertices).
   For every edge u -> v with weight w, update dist[v] = min(dist[v], dist[u] + w).

3. Check for Negative Weight Cycles:
   After the V-1 iterations, check all edges again.
   If any edge can still be relaxed, a negative weight cycle exists.

```
def bellman_ford(graph, V, src):

    dist = [float('inf')] * V

    dist[src] = 0

    for _ in range(V - 1):

        for u, v, w in graph:

            if dist[u] != float('inf') and dist[u] + w < dist[v]:

                dist[v] = dist[u] + w

    for u, v, w in graph:

        if dist[u] != float('inf') and dist[u] + w < dist[v]:

            return "Graph contains a negative weight cycle"



    return dist
```

Complexity:

- **Time Complexity**: O(V×E), where E is the number of edges.
- **Space Complexity**: O(V)

Comparison:

Algorithm	Use Case	Handles Negative Weights	Detects Negative Cycles	Time Complexity	Space Complexity
Floyd-Warshall	All-pairs shortest paths	Yes	Yes	O(V^3)	O(V^2)
Bellman-Ford	Single-source shortest path	Yes	Yes	O(V×E)	O(V)