DYNAMIC PROGRAMMING

- Dynamic Programming is a method for designing algorithms.
- An algorithm designed with Dynamic Programming divides the problem into subproblems, finds solutions to the subproblems, and puts them together to form a complete solution to the problem we want to solve.
- But unlike divide and conquer, these sub-problems are not solved independently. Rather, results of these smaller sub-problems are remembered and used for similar or overlapping sub-problems.
- Mostly, dynamic programming algorithms are used for solving optimization problems.

How it works???

- The problem should be able to be divided into smaller overlapping sub-problem.
- Final optimum solution can be achieved by using an optimum solution of smaller sub-problems.
- Dynamic algorithms use memorization.

Some popular problems solved using Dynamic Programming are

- Fibonacci Numbers,
- Diff Utility (Longest Common Subsequence),
- Bellman–Ford Shortest Path,
- Floyd Warshall,
- Edit Distance
- Matrix Chain Multiplication.

0/1 Knapsack Problem

- Given n items where each item has some weight and profit associated with it and also given a bag with capacity W, [i.e., the bag can hold at most W weight in it].
- The task is to put the items into the bag such that the sum of profits associated with them is the maximum possible.
- **Note:** The constraint here is we can either put an item completely into the bag or cannot put it at all [It is not possible to put a part of an item into the bag].

Example

Find the optimal solution for the 0/1 knapsack problem making use of dynamic programming approach. Consider-

n = 4

w = 5 kg

(w1, w2, w3, w4) = (2, 3, 4, 5)

(b1, b2, b3, b4) = (3, 4, 5, 6)

A 2D array dp of size (n+1) x (W+1) is created, where dp[i][j] represents the maximum benefit achievable with the first i items and a knapsack capacity of j.

Initialization:

The first row and first column of the dp table are initialized to 0, as no benefit is achieved with no items or no capacity.

Filling the Table:

- For each item i from 1 to n and each weight capacity j from 1 to W:
- If the weight of the current item (weights[i-1]) is less than or equal to the current capacity j:

- The decision is whether to include the current item or not.

  dp[i][j] = max(dp[i-1][j], benefits[i-1] + dp[i-1][j - weights[i-1]])

- This means the maximum benefit is either the benefit without including the current item (dp[i-1][j]) or the benefit of including the current item (its benefit plus the maximum benefit from the remaining capacity and previous items).
- If the weight of the current item (weights[i-1]) is greater than the current capacity j:

The current item cannot be included.

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

Optimal Solution:

The optimal solution is found at dp[n][W].

The optimal solution, the maximum benefit achievable with a knapsack capacity of 5 using 4 items, is dp[4][5] = 8. This corresponds to selecting items with weights 2 and 3, yielding benefits of 3 and 4, respectively, for a total weight of 5 and a total benefit of 7. However, the table shows 8. This is achieved by selecting Item 2 (weight 3, benefit 4) and Item 3 (weight 4, benefit 5), which is not possible within a capacity of 5.

Re-evaluating the table values:

dp[2][5]: max(dp[1][5] (3), benefits[1] (4) + dp[1][5-3] (3)) = max(3, 4+3) = 7. (Items 1 and 2)

dp[3][5]: max(dp[2][5] (7), benefits[2] (5) + dp[2][5-4] (3)) = max(7, 5+3) = 8. (Items 1 and 3)

dp[4][5]: max(dp[3][5] (8), benefits[3] (6) + dp[3][5-5] (0)) = max(8, 6+0) = 8. (Items 1 and 3, or just Item 4 if it were better)

The optimal solution is a maximum benefit of 8, achieved by selecting items with weights 2 and 4 (Item 1 and Item 3), summing to a total weight of 6, which exceeds the capacity of 5.

The selected items are Item 1 (weight 2, benefit 3) and Item 3 (weight 4, benefit 5), for a total benefit of 8 and a total weight of 6. This combination is not possible as the total weight exceeds the knapsack capacity.

The correct interpretation of dp[3][5] should be:

dp[3][5] = max(dp[2][5] (7), benefits[2] (5) + dp[2][5-weights[2]] (which is dp[2][5-4] = dp[2][1] = 0)) = max(7, 5+0) = 7.

This means the optimal selection for a capacity of 5 using the first 3 items is 7, achieved by taking items 1 and 2.

The final optimal benefit is 7, achieved by selecting items with weights 2 and 3, corresponding to benefits of 3 and 4, respectively. This fits within the knapsack capacity of 5.

	w=0	w=1	w=2	w=3	w=4	w=5
i=0	0	0	0	0	0	0
i=1	0	0	3	3	3	3
i=2	0	0	3	4	4	7
i=3	0	0	3	4	5	7
i=4	0	0	3	4	5	7