Introduction of Dynamic Programming

Dynamic Programming is the most powerful design technique for solving optimization problems.

Divide & Conquer algorithm partition the problem into disjoint subproblems solve the subproblems recursively and then combine their solution to solve the original problems.

Dynamic Programming is used when the subproblems are not independent, e.g. when they share the same subproblems. In this case, divide and conquer may do more work than necessary, because it solves the same sub problem multiple times.

Dynamic Programming solves each subproblems just once and stores the result in a table so that it can be repeatedly retrieved if needed again.

Dynamic Programming is a **Bottom-up approach-** we solve all possible small problems and then combine to obtain solutions for bigger problems.

Dynamic Programming is a paradigm of algorithm design in which an optimization problem is solved by a combination of achieving sub-problem solutions and appearing to the "**principle of optimality**".

Characteristics of Dynamic Programming:

Dynamic Programming works when a problem has the following features:-

- **Optimal Substructure:** If an optimal solution contains optimal sub solutions then a problem exhibits optimal substructure.
- **Overlapping subproblems:** When a recursive algorithm would visit the same subproblems repeatedly, then a problem has overlapping subproblems.

If a problem has optimal substructure, then we can recursively define an optimal solution. If a problem has overlapping subproblems, then we can improve on a recursive implementation by computing each subproblem only once.

If a problem doesn't have optimal substructure, there is no basis for defining a recursive algorithm to find the optimal solutions. If a problem doesn't have overlapping sub problems, we don't have anything to gain by using dynamic programming.

If the space of subproblems is enough (i.e. polynomial in the size of the input), dynamic programming can be much more efficient than recursion.

Elements of Dynamic Programming

There are basically three elements that characterize a dynamic programming algorithm:-

1. **Substructure:** Decompose the given problem into smaller subproblems. Express the solution of the original problem in terms of the solution for smaller problems.
2. **Table Structure:** After solving the sub-problems, store the results to the sub problems in a table. This is done because subproblem solutions are reused many times, and we do not want to repeatedly solve the same problem over and over again.
3. **Bottom-up Computation:** Using table, combine the solution of smaller subproblems to solve larger subproblems and eventually arrives at a solution to complete problem.

Note: Bottom-up means:-

i. Start with smallest subproblems.
ii. Combining their solutions obtain the solution to sub-problems of increasing size.
iii. Until solving at the solution of the original problem.

Components of Dynamic programming

1. **Stages:** The problem can be divided into several subproblems, which are called stages. A stage is a small portion of a given problem. For example, in the shortest path problem, they were defined by the structure of the graph.
2. **States:** Each stage has several states associated with it. The states for the shortest path problem was the node reached.
3. **Decision:** At each stage, there can be multiple choices out of which one of the best decisions should be taken. The decision taken at every stage should be optimal; this is called a stage decision.
4. **Optimal policy:** It is a rule which determines the decision at each stage; a policy is called an optimal policy if it is globally optimal. This is known as Bellman principle of optimality.
5. Given the current state, the optimal choices for each of the remaining states does not depend on the previous states or decisions. In the shortest path problem, it was not necessary to know how we got a node only that we did.
6. There exist a recursive relationship that identify the optimal decisions for stage j, given that stage j+1, has already been solved.
7. The final stage must be solved by itself.

Development of Dynamic Programming Algorithm

It can be broken into four steps:

1. Characterize the structure of an optimal solution.
2. Recursively defined the value of the optimal solution. Like Divide and Conquer, divide the problem into two or more optimal parts recursively. This helps to determine what the solution will look like.
3. Compute the value of the optimal solution from the bottom up (starting with the smallest subproblems)
4. Construct the optimal solution for the entire problem form the computed values of smaller subproblems.

Applications of dynamic programming

1. 0/1 knapsack problem
2. Mathematical optimization problem
3. All pair Shortest path problem
4. Reliability design problem
5. Longest common subsequence (LCS)
6. Flight control and robotics control
7. Time-sharing: It schedules the job to maximize CPU usage

0/1 Knapsack Problem: Dynamic Programming Approach:

Knapsack Problem:

Knapsack is basically means bag. A bag of given capacity.

We want to pack n items in your luggage.

- The ith item is worth v_i dollars and weight w_i pounds.
- Take as valuable a load as possible, but cannot exceed W pounds.
- v_i w_i W are integers.

1. W ≤ capacity
2. Value ← Max

- Knapsack of capacity
- List (Array) of weight and their corresponding value.

Output: To maximize profit and minimize weight in capacity.

The knapsack problem where we have to pack the knapsack with maximum value in such a manner that the total weight of the items should not be greater than the capacity of the knapsack.

Knapsack problem can be further divided into two parts:

1. Fractional Knapsack: Fractional knapsack problem can be solved by **Greedy Strategy** where as 0 /1 problem is not.

It cannot be solved by **Dynamic Programming Approach.**

0/1 Knapsack Problem:

In this item cannot be broken which means thief should take the item as a whole or should leave it. That's why it is called **0/1 knapsack Problem**.

- Each item is taken or not taken.
- Cannot take a fractional amount of an item taken or take an item more than once.
- It cannot be solved by the Greedy Approach because it is enable to fill the knapsack to capacity.
- **Greedy Approach** doesn't ensure an Optimal Solution.

Example of 0/1 Knapsack Problem:

Example: The maximum weight the knapsack can hold is W is 11. There are five items to choose from. Their weights and values are presented in the following table:

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1\ v_1 = 1$												
$w_2 = 2\ v_2 = 6$												
$w_3 = 5\ v_3 = 18$												
$w_4 = 6\ v_4 = 22$												
$w_5 = 7\ v_5 = 28$												

The [i, j] entry here will be V [i, j], the best value obtainable using the first "i" rows of items if the maximum capacity were j. We begin by initialization and first row.

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1\ v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2\ v_2 = 6$	0											
$w_3 = 5\ v_3 = 18$	0											
$w_4 = 6\ v_4 = 22$	0											
$w_5 = 7\ v_5 = 28$	0											

+

+

```
V [i, j] = max {V [i - 1, j], vi + V [i - 1, j -wi]
```

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1\ v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2\ v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5\ v_3 = 18$	0											
$w_4 = 6\ v_4 = 22$	0											
$w_5 = 7\ v_5 = 28$	0											

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1\ v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2\ v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5\ v_3 = 18$	0	1	6	7	7	18	19	24	25	25	25	25
$w_4 = 6\ v_4 = 22$	0											
$w_5 = 7\ v_5 = 28$	0											

The value of V [3, 7] was computed as follows:

```
V [3, 7] = max {V [3 - 1, 7], v3 + V [3 - 1, 7 - w3]
         = max {V [2, 7], 18 + V [2, 7 - 5]}
         = max {7, 18 + 6}
         = 24
```

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1\ v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2\ v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5\ v_3 = 18$	0	1	6	7	7	18	19	24	25	25	25	25
$w_4 = 6\ v_4 = 22$	0	1	6	7	7	18	22	24	28	29	29	40
$w_5 = 7\ v_5 = 28$	0											

Finally, the output is:

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1\ v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2\ v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5\ v_3 = 18$	0	1	6	7	7	18	19	24	25	25	25	25
$w_4 = 6\ v_4 = 22$	0	1	6	7	7	18	22	24	28	29	29	40
$w_5 = 7\ v_5 = 28$	0	1	6	7	7	18	22	28	29	34	35	40

The maximum value of items in the knapsack is 40, the bottom-right entry). The dynamic programming approach can now be coded as the following algorithm:

Algorithm of Knapsack Problem

```
KNAPSACK (n, W)
 1. for w = 0, W
 2. do V [0,w] ← 0
 3. for i=0, n
 4. do V [i, 0] ← 0
 5. for w = 0, W
 6. do if (wi≤ w & vi + V [i-1, w -  wi]> V [i -1,W])
 7. then V [i, W] ← vi + V [i - 1, w - wi]
 8. else V [i, W] ← V [i - 1, w]
```