Assignment No. 6

Aim: Implement a program in C++ for 0/1 Knapsack problem using Dynamic Programming method.

Objectives:
1. To understand the Dynamic Programming through 0/1 Knapsack Problem

Theory:

- **What is Dynamic Programming?**

  Dynamic Programming is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming. The idea is to simply store the results of subproblems, so that we do not have to re-compute them when needed later. This simple optimization reduces time complexities from exponential to polynomial.

  For example, if we write simple recursive solution for Fibonacci Numbers, we get exponential time complexity and if we optimize it by storing solutions of subproblems, time complexity reduces to linear.

- **What is Knapsack's Problem:**

  Given weights and values of N items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays value [0..N-1] and weight [0..N-1] which represent values and weights associated with N items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that the sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item or don't pick it (0-1 property).

- **For Example:**

  Input: N = 3, W = 4

  values[] = {1,2,3}

  weight[] = {4,5,1}

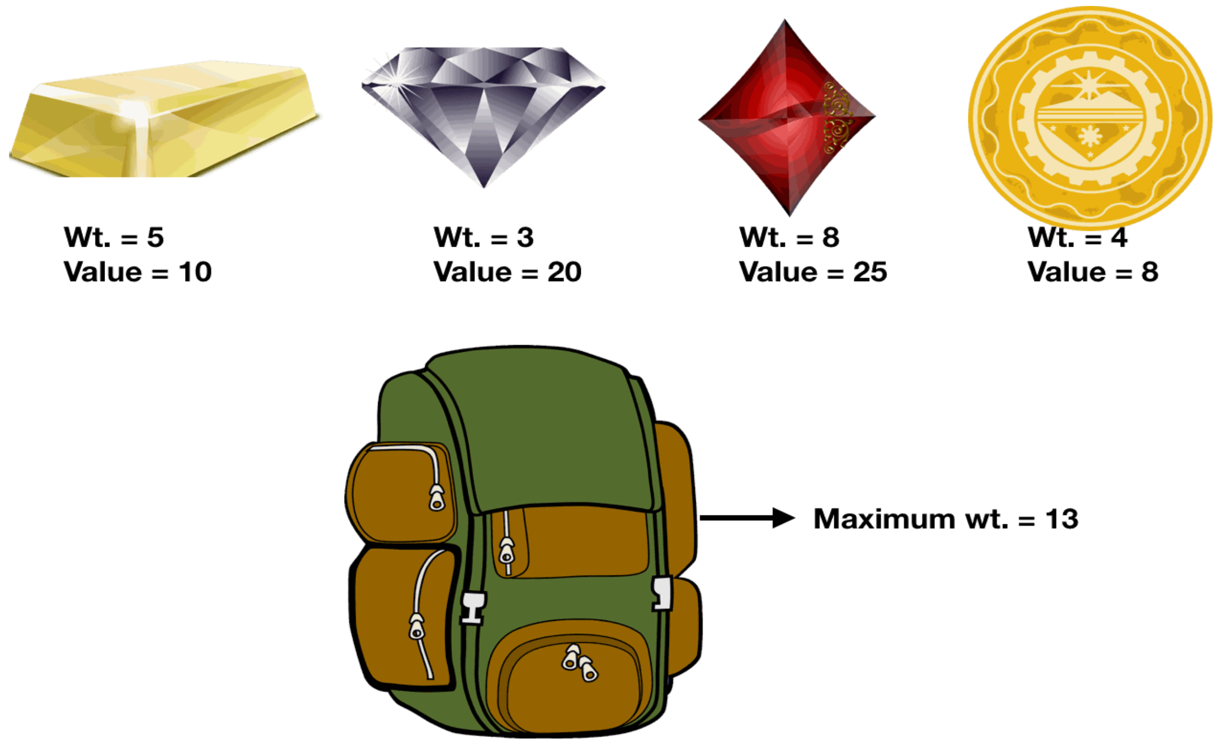
  Output: 3

  Input: N = 3, W = 3

  values[] = {1,2,3}

  weight[] = {4,5,6}

  Output: 0



- **Solution for Knapsack's Problem using Dynamic Programming:**

  Since subproblems are evaluated again, this problem has Overlapping Sub-problems property. So the 0-1 Knapsack problem has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, re-computation of the same subproblems can be avoided by constructing a temporary array K[][] in a bottom-up manner.

Follow the below steps to solve the problem:

Consider the same cases as mentioned in the recursive approach.

- In a DP[][] table let's consider all the possible weights from '1' to 'W' as the columns and weights that can be kept as rows.
- The state DP[i][j] will denote the maximum value of 'j-weight' considering all values from '1 to ith'. So if we consider 'wi' (weight in 'ith' row) we can fill it in all columns which have 'weight values > wi'. Now two possibilities can take place:
  - Fill 'wi' in the given column.
  - Do not fill 'wi' in the given column.
  - Now we have to take a maximum of these two possibilities, formally if we do not fill the 'ith' weight in the 'jth' column then the DP[i][j] state will be the same as DP[i-1][j] but if we fill the weight, DP[i][j] will be equal to the value of 'wi'+ value of the column weighing 'j-wi' in the previous row.
  - So, we take the maximum of these two possibilities to fill the current state

- Let weight elements = {1, 2, 3}
  Let weight values = {10, 15, 40}
  Capacity=6

-

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0	10	15	25	25	25	25

- 3 0

- **Explanation:**
- For filling ‘weight = 2’ we come
  across ‘j = 3’ in which
  we take maximum of
  (10, 15 + DP[1][3-2]) = 25
  | |
  ‘2 not filled’   ‘2 filled’

-

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0	10	15	25	25	25	25
3	0	10	15	40	50	55	65

- **Explanation:**
- For filling ‘weight=3’,
  we come across ‘j=4’ in which
  we take maximum of (25, 40 + DP[2][4-3]) = 50

- For filling ‘weight=3’
  we come across ‘j=5’ in which
  we take maximum of (25, 40 + DP[2][5-3]) = 55

- For filling ‘weight=3’
  we come across ‘j=6’ in which
  we take maximum of (25, 40 + DP[2][6-3]) = 65

Program:

Output:

Conclusion: