Experiment No. 10

Title: Write a program to find solution to 0 / 1 Knapsack Problem Instance.

Theory:

A knapsack is a bag. The knapsack problem deals with the putting items to the bag based on the value of the items. It aim is to maximise the value inside the bag. In 0-1 Knapsack you can either put the item or discard it, there is no concept of putting some part of item in the knapsack.

Like other typical Dynamic Programming(DP) problems, re-computation of same subproblems can be avoided by constructing a temporary array K[][] in bottom-up manner. Following is Dynamic Programming based implementation.

Approach: In the Dynamic programming we will work considering 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 the rows.
The state DP[i][j] will denote 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 'ith' weight in 'jth' column then DP[i][j] state will be 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. This visualisation will make the concept clear:

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'   '2 filled'
not 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

Algorithm

```
max(int a, int b)
{
    if (a > b)
      return a
    else
      return b
}

knapSack(W, wt[],val[],n)
{
 //Two dimensional array K[n+1][W+1]
    // Build table K[][] in bottom up manner
    for(i = 0; i <= n; i++)
    {
        for(w = 0; w <= W; w++)
        {
            if (i == 0 || w == 0)
                K[i][w] = 0;
            else if (wt[i - 1] <= w)
                K[i][w] = max(val[i - 1] +
                                K[i - 1][w - wt[i - 1]],
                                K[i - 1][w]);
            else
                K[i][w] = K[i - 1][w];
        }
    }
    return K[n][W];
}
```

Complexity Analysis:

Time Complexity: O(N*W).
where ‘N’ is the number of weight element and ‘W’ is capacity. As for every weight element we traverse through all weight capacities 1<=w<=W.

Auxiliary Space: O(N*W).
The use of 2-D array of size ‘N*W’.