**Practical – 04**

**Aim :**

Write a program to solve a 0-1 Knapsack problem using dynamic programming or branch and bound strategy.

**Program:**

public class Knapsack {

public static void main(String[] args) {

int capacity = 10;

int items = 4;

int[] price = {0, 3, 7, 2, 9};

int[] wt = {0, 2, 2, 4, 5};

int[][] dp = new int[items + 1][capacity + 1];

for (int i = 0; i <= items; i++) {

for (int j = 0; j <= capacity; j++) {

if (i == 0 || j == 0) {

// There's nothing to add to the knapsack

dp[i][j] = 0;

} else if (wt[i] <= j) {

// Choose previously maximum or value of the current item + value of remaining weight

dp[i][j] = Math.max(dp[i - 1][j], price[i] + dp[i - 1][j - wt[i]]);

} else {

// Add previously added item to the knapsack

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

}

}

}

System.out.println("Maximum Profit Earned: " + dp[items][capacity]);

}

}

/\*

0/1 Knapsack :

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).

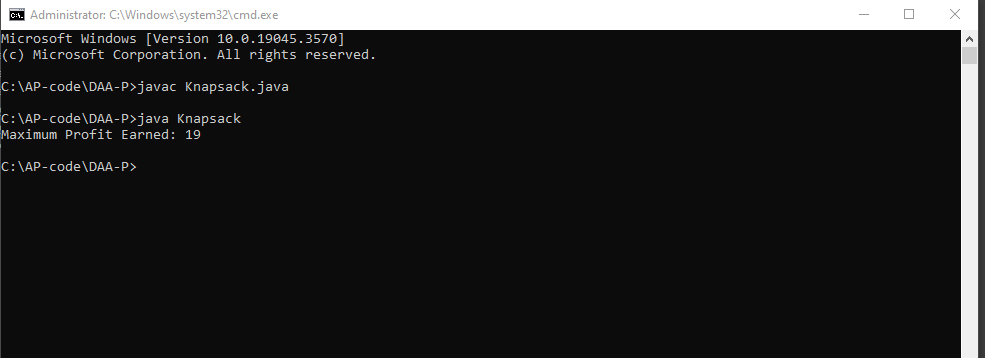
The use of 2-D array of size ‘N\*W’.

Knapsack problem using dynamic programming or branch and

bound strategy.

\*/

**Output:**

****