\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Assignment No: 5**

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

**Name**: **Prasanna Satish Kulkarni**

**Class**: **BE** **Div: BE-I** **Batch**: **B**

**Exam Seat No/Roll No: 405A048**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Program:**

using namespace std;

#include<iostream>

// A utility function that returns

// maximum of two integers

int max(int a, int b) { return (a > b) ? a : b; }

// Returns the maximum value that

// can be put in a knapsack of capacity W

int knapSack(int W, int wt[], int val[], int n)

{

// Base Case

if (n == 0 || W == 0)

return 0;

// If weight of the nth item is more

// than Knapsack capacity W, then

// this item cannot be included

// in the optimal solution

if (wt[n - 1] > W)

return knapSack(W, wt, val, n - 1);

// Return the maximum of two cases:

// (1) nth item included

// (2) not included

else

return max(

val[n - 1]

+ knapSack(W - wt[n - 1],

wt, val, n - 1),

knapSack(W, wt, val, n - 1));

}

int main()

{

int val[] = { 60, 100, 120 };

int wt[] = { 10, 20, 30 };

int W = 50;

int n = sizeof(val) / sizeof(val[0]);

std::cout << knapSack(W, wt, val, n);

return 0;

}

**Output:**

**220**