

|  |
| --- |
| **Title: Implementation of Knapsack Problem using Dynamic Programming** |



**Objective** To learn the Dynamic Programming using Knapsack Problemalgorithm

**CO to be achieved:**

|  |  |
| --- | --- |
| Sr. No | Objective |
| CO 1 | Compare and demonstrate the efficiency of algorithms using asymptotic complexity notations. |
| CO 2 | Analyze and solve problems for divide and conquer strategy, greedy method, dynamic programming approach and backtracking and branch & bound policies. |
| CO 3 | Analyze and solve problems for   different string matching algorithms. |



**Books/ Journals/ Websites referred:**

1. **Ellis horowitz, Sarataj Sahni, S.Rajsekaran,” Fundamentals of computer algorithm”, University Press**
2. **T.H.Cormen ,C.E.Leiserson,R.L.Rivest and C.Stein,” Introduction to algortihtms”,2nd Edition ,MIT press/McGraw Hill,2001**
3. **en.wikipedia.org/wiki/Knapsack\_problem**
4. **www.es.ele.tue.nl/education/5MC10/Solutions/knapsack.pdf**
5. **cse.unl.edu/~ylu/raik283/notes/0-1-knapsack.ppt**
6. **www.es.ele.tue.nl/education/5MC10/Solutions/knapsack.pdf**
7. **cse.unl.edu/~ylu/raik283/notes/0-1-knapsack.ppt**



**Pre Lab/ Prior Concepts:**

Data structures, Concepts of algorithm analysis



**Historical Profile:**

Dynamic Programming (DP) is used heavily in optimization problems (finding the maximum and the minimum of something). Applications range from financial models and operation research to biology and basic algorithm research. So the good news is that understanding DP is profitable. However, the bad news is that DP is not an algorithm or a data structure that you can memorize. It is a powerful algorithmic design technique.

**Principle of Optimality:** It states that an optimal sequence of decisions has the property that whatever the initial state and decision are, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision.

The solution to the knapsack problem can be viewed as the result of a sequence of decisions. We have to decide the values of xi, 1<= i<+n. First we make a decision on xi , then on x2 . then on x3 ,etc. An optimal sequence of decisions maximizes the objective function ∑pixi.



**New Concepts to be learned:**

Application of algorithmic design strategy to any problem, dynamic Programming method of problem solving Vs other methods of problem solving, optimality of the solution,



**Algorithm**

// Input:

// Values (stored in array v)

// Weights (stored in array w)

// Number of distinct items (n)

// Knapsack capacity (W)

for j from 0 to W do

m[0, j] := 0

end for

for i from 1 to n do

for j from 0 to W do

if w[i] <= j then

m[i, j] := max(m[i-1, j], m[i-1, j-w[i]] + v[i])

else

m[i, j] := m[i-1, j]

end if

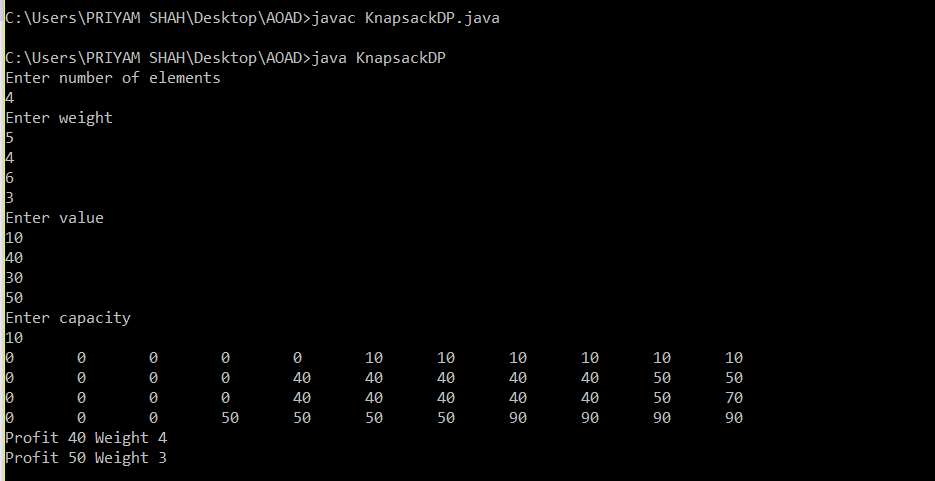
end for

end for

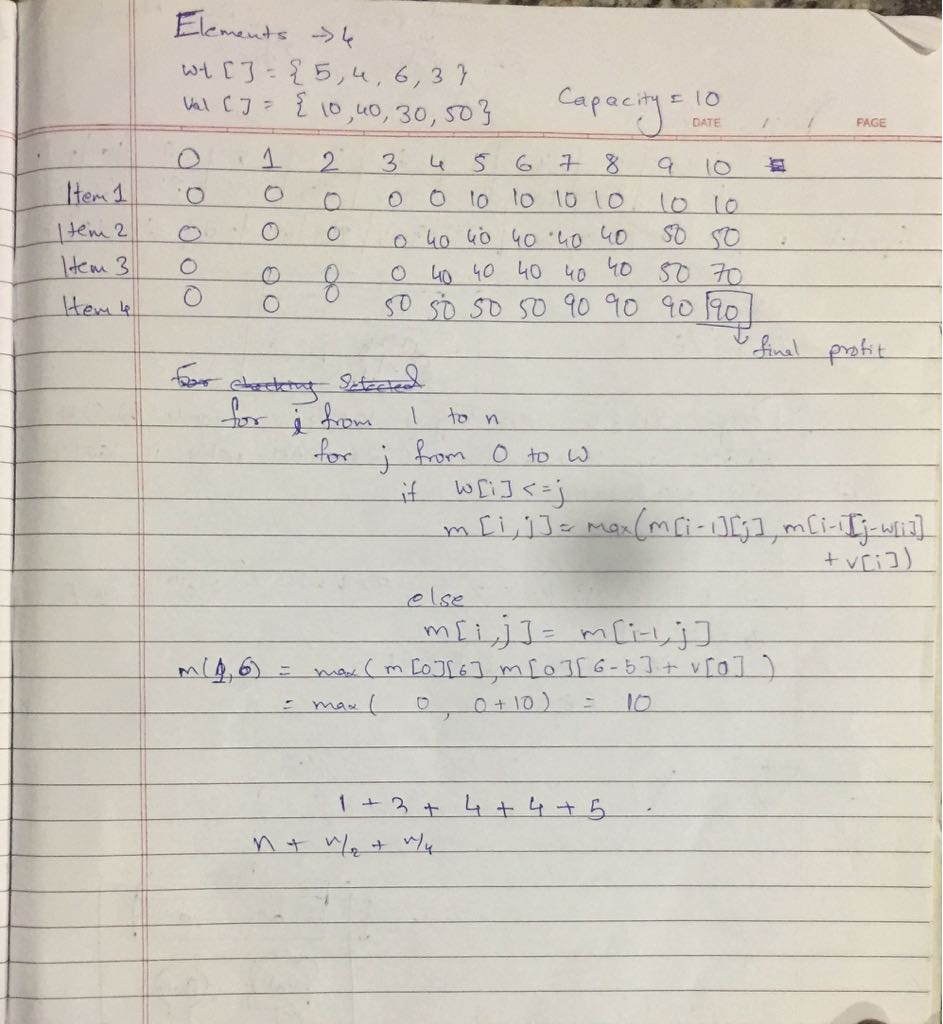
**Code:**

**import java.util.Scanner;  
   
public class KnapsackDP  
{  
 public void sol(int[] wt, int[] val, int W, int N)  
 {  
   
 int[][] m = new int[N + 1][W + 1];  
 int[][] sol = new int[N + 1][W + 1];  
 for (int i = 1; i <= N; i++)  
 {  
 for (int j = 0; j <= W; j++)  
 {  
   
 if (j >= wt[i]){  
 m[i][j] = Math.max(m[i - 1][j], m[i - 1][j - wt[i]] + val[i]);  
 System.out.print(m[i][j]+"\t");  
 sol[i][j] = (m[i - 1][j - wt[i]] + val[i]) > (m[i - 1][j]) ? 1 : 0;}  
 else{  
 m[i][j]= m[i-1][j];  
 System.out.print(m[i][j]+"\t");  
 }  
   
 }  
 System.out.println();  
 }   
 int[] selected = new int[N + 1];  
 for (int n = N, w = W; n > 0; n--)  
 {  
 if (sol[n][w] != 0)  
 {  
 selected[n] = 1;  
 w = w - wt[n];  
 }  
 else  
 selected[n] = 0;  
 }  
 for (int i = 1; i < N + 1; i++)  
 if (selected[i] == 1){  
 System.out.print("Profit "+val[i] +" ");  
 System.out.print("Weight "+wt[i]+" \n");  
 }  
   
 }  
 public static void main (String[] args)   
 {  
 Scanner sc = new Scanner(System.in);  
 KnapsackDP ks = new KnapsackDP();  
   
 System.out.println("Enter number of elements ");  
 int n = sc.nextInt();  
   
 int[] wt = new int[n + 1];  
 int[] val = new int[n + 1];  
   
 System.out.println("Enter weight");  
 for (int i = 1; i <= n; i++)  
 wt[i] = sc.nextInt();  
 System.out.println("Enter value ");  
 for (int i = 1; i <= n; i++)  
 val[i] = sc.nextInt();  
   
 System.out.println("Enter capacity ");  
 int W = sc.nextInt();  
   
 ks.sol(wt, val, W, n);  
 }  
}**

**Output:**

****

**Example:**

****

**Analysis of 0/1 Knapsack algorithm:**

**0-1 Knapsack cannot be solved by Greedy approach. Greedy approach does not ensure an optimal solution.**

**This algorithm takes θ(*n*, *w*) times as table *c* has (*n* + 1).(*w* + 1) entries, where each entry requires θ(1) time to compute.**

**CONCLUSION:**

**Implementation of Knapsack Problem using Dynamic Programming has been studied.**