**Topic: DYNAMIC PROGRAMMING**

**Theory:** Dynamic programming is an algorithm design method that can be used when the solution to a problem can be viewed as the result of a sequence of decisions. For some of the problems that may be viewed in this way ,an optimal sequence of decisions can be found by making the decisions one at a time and never making an erroneous decision .This is true for all problem solvable by the greedy method. For many other problems, it is not possible to make stepwise decisions in such a manner that the sequence of decisions made is optimal.

**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.

**Example 1[knapsack]:**

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.

**Example 2 [Optimal merge patterns]:**

An optimal merge pattern tells us which pair of files should be merged at each step. As a decision sequence, the problem calls for us to decide which pair of files should be merged first, which pair second, which pair third, etc. An optimal sequence of decisions is a least-cost sequence.

**Batch: B1 Roll No.: 1611077**

**Experiment No. \_\_6\_\_**

**Grade: AA / AB / BB / BC / CC / CD /DD**

**Signature of the Staff In-charge with date**

|  |
| --- |
| **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

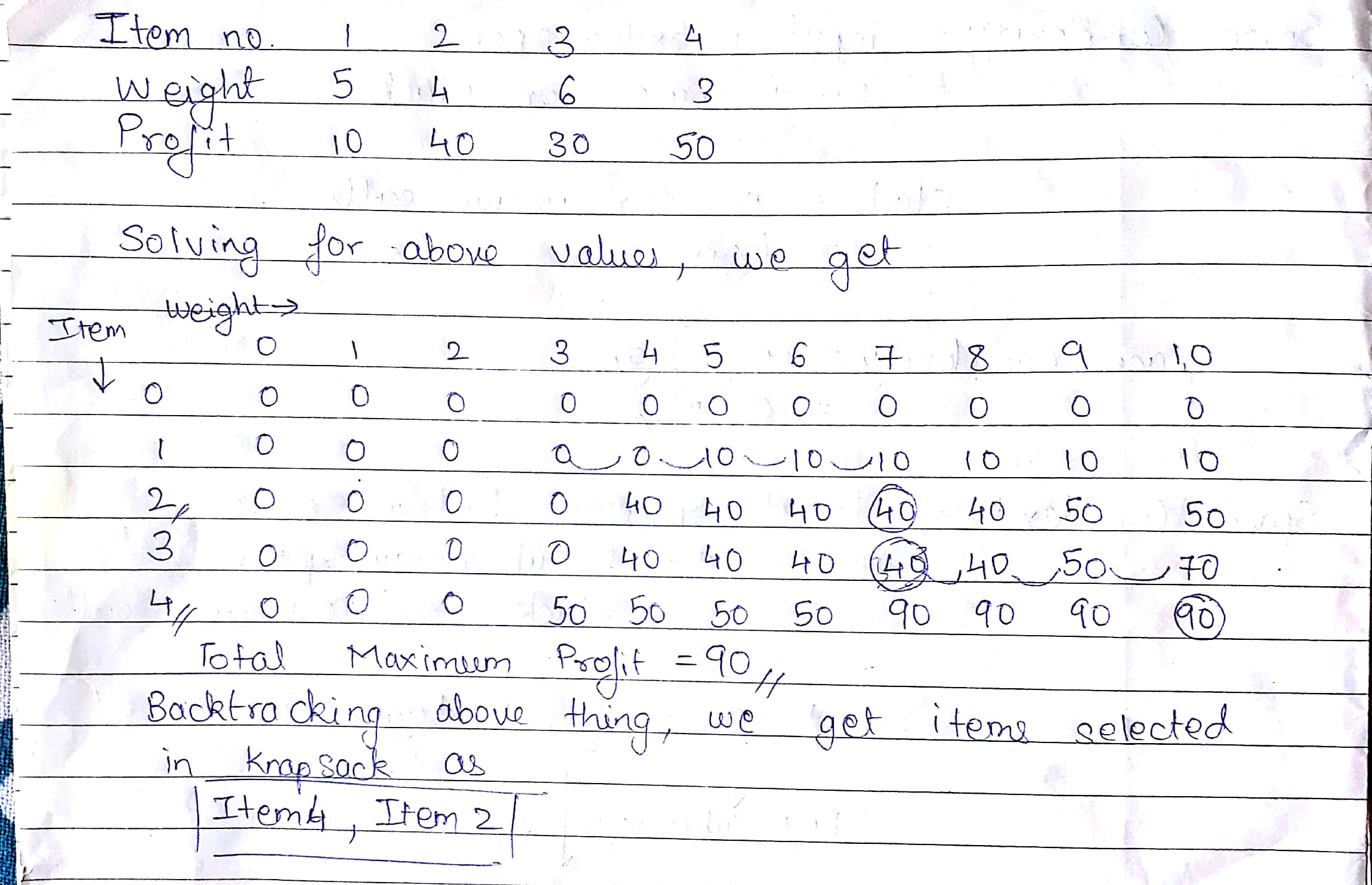
**Example:**

n=4, W=10

w[]={5,4,6,3}

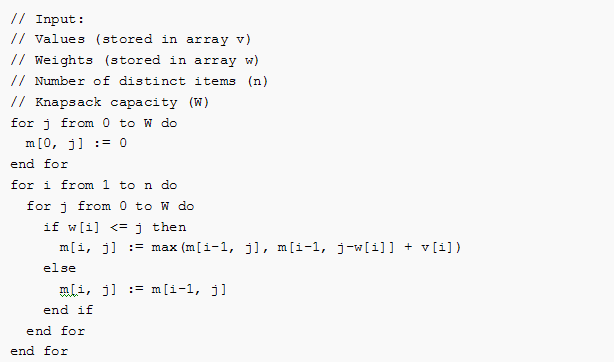
P[]={10,40,30,50}

**Solution:**

****

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

Algorithm:

****

* In the above algorithm, outer loop runs n times and inner loop runs W times where, n is the number of items and W is the capacity of knapsack.
* Therefore, Time Complexity of the algorithm is O(nW).

**IMPLEMENTATION**

import java.util.\*;

class Knap

{

public static void main(String[] args)

{

Scanner sc=new Scanner(System.in);

int i,j;

System.out.println("Enter the max. weight the knapsack can carry: ");

int w=sc.nextInt();

System.out.println("Enter the no. of items: ");

int n=sc.nextInt();

System.out.println("Enter the weights and profits: ");

int a[][]=new int[n][];

int b[][]=new int[n+1][];

for(i=0;i<n;i++)

{

a[i]=new int[2];

for(j=0;j<2;j++)

{

a[i][j]=sc.nextInt();

}

}

b[0]=new int[w+1];

for(i=1;i<=n;i++)

{

b[i]=new int[w+1];

for(j=1;j<=w;j++)

{

if(a[i-1][0]>j)

{

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

}

else

{

b[i][j]=Math.max(b[i-1][j],(b[i-1][j-(a[i-1][0])])+a[i-1][1]);

}

}

}

for(i=0;i<=n;i++)

{

for(j=0;j<=w;j++)

{

System.out.print(b[i][j]+" ");

}

System.out.print("\n");

}

System.out.println("Profit is "+b[n][w]);

j=n;

i=w;

while(j>=1)

{

if(b[j][i]!=b[j-1][i])

{

System.out.println("Object "+j+" selected");

i=i-a[j-1][0];

}

j--;

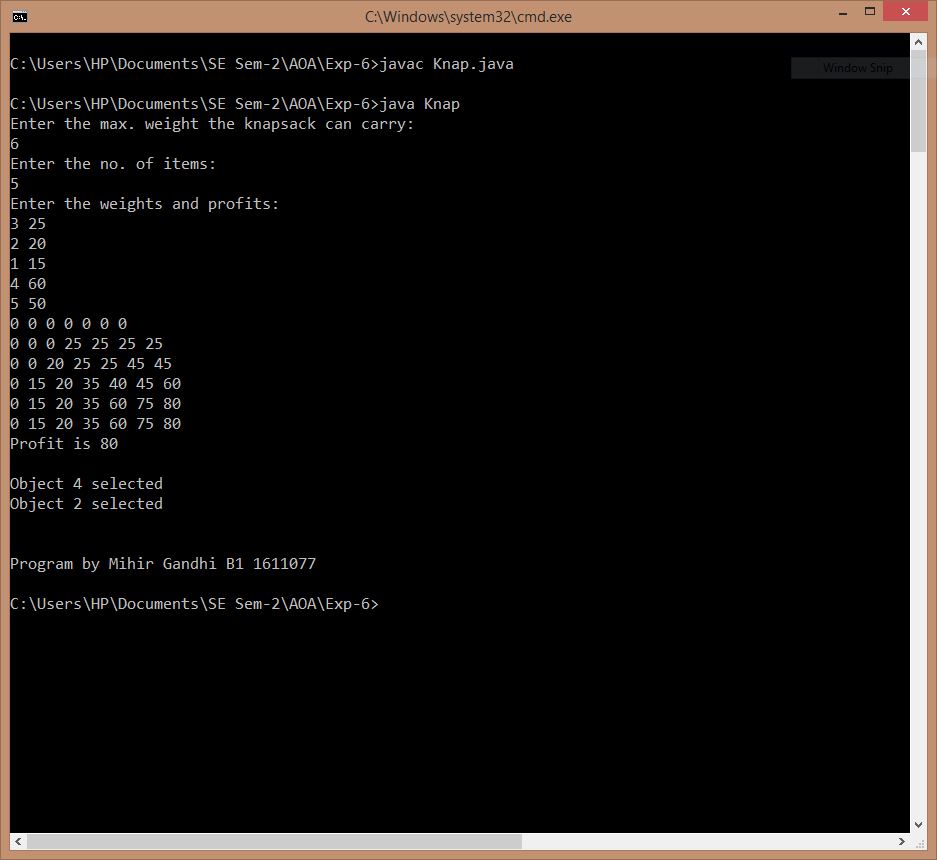
}

System.out.print("\n\nProgram by Mihir Gandhi B1 1611077 \n");

}

}

**OUTPUT:**

****

**CONCLUSION**

Thus, 0-1 Knapsack Problem has been successfully implemented using Dynamic Programming.

The actual outcome matched with the expected outcome.