0/1 knapsack problem

Design and Analysis of Algorithms Project Based

Lab Report

Bachelor of Technology
in

Department of Electronics and Computer Engineering

By

Bharadwaj 180050068

under the supervision of

Dr. M. Ramesh Kumar

Assistant Professor, Ph.D.

(DEEMED TO BE UNIVERSITY)

Koneru Lakshmaiah Education Foundation

(Deemed to be University estd., u/s 3 of UGC Act 1956)

Greenfields, Vaddeswaram, Guntur (Dist.), Andhra Pradesh - 522502

April 2021

KLEF

DEPARTMENT OF ELECTRONICS&COMPUTER SCIENCE ENGINEERING

This is to certify that this project-based lab report entitled “**Knapsack problem**” is a Bonafide work done by Bharadwaj (180050068) in partial fulfillment of the requirement for the award of degree in BACHELOR OF TECHNOLOGY in Electronic &Computer Science Engineering during the academic year 2020-2021.

Professor in Charge	**Head of the Department**
Dr. M. Ramesh Kumar	**Dr. M S D Prasad**

ACKNOWLEDGEMENT

Our sincere thanks to **Dr. M. Ramesh Kumar** in the Lab for their outstanding support throughout the project for the successful complete on of the work.

We express our gratitude to **Dr. M S G PRASAD,** Head of the Department for Electronics &Computer Science Engineering for providing us with adequate facilities, ways and means by which we can complete this project work.

We would like to place on record the deep sense of gratitude to the honorable Vice Chancellor, K L University for providing the necessary facilities to carry the concluded project work.

Last but not the least, we thank all Teaching and Non-Teaching Staff of our department and especially my classmates and my friends for their support in the completion of our project work.

Bharadwaj 180050068

Table of Contents

1. Introduction

Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item or don't pick it (0-1 property).

2. Problem Description

Knapsack is basically means bag. A bag of given capacity.
We want to pack n items in your luggage.
The ith item is worth vi dollars and weight wi pounds.
Take as valuable a load as possible, but cannot exceed W pounds.
vi wi W are integers.

3. Techniques /Approaches for solving the problem.

Method 1:

A simple solution is to consider all subsets of items and calculate the total weight and value of all subsets. Consider the only subsets whose total weight is smaller than W. From all such subsets, pick the maximum value subset..

Method 2:

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 visualization will make the concept clear: .

4.Examples

The maximum weight the knapsack can hold is W is 11. There are five items to choose from. Their weights and values are presented in the following table:

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1$ $v_1 = 1$												
$w_2 = 2$ $v_2 = 6$												
$w_3 = 5$ $v_3 = 18$												
$w_4 = 6$ $v_4 = 22$												
$w_5 = 7$ $v_5 = 28$												

The [i, j] entry here will be V [i, j], the best value obtainable using the first "i" rows of items if the maximum capacity were j. We begin by initialization and first row.

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1$ $v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2$ $v_2 = 6$	0											
$w_3 = 5$ $v_3 = 18$	0											
$w_4 = 6$ $v_4 = 22$	0											
$w_5 = 7$ $v_5 = 28$	0											

$V[i, j] = \max \{V[i - 1, j], v_i + V[i - 1, j - w_i]$

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1$ $v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2$ $v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5$ $v_3 = 18$	0											
$w_4 = 6$ $v_4 = 22$	0											
$w_5 = 7$ $v_5 = 28$	0											

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1$ $v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2$ $v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5$ $v_3 = 18$	0	1	6	7	7	18	19	24	25	25	25	25
$w_4 = 6$ $v_4 = 22$	0											
$w_5 = 7$ $v_5 = 28$	0											

The value of V [3, 7] was computed as follows:

$$
\begin{aligned}
V[3, 7] &= \max \{V[3 - 1, 7], v_3 + V[3 - 1, 7 - w_3]\\
&= \max \{V[2, 7], 18 + V[2, 7 - 5]\}\\
&= \max \{7, 18 + 6\}\\
&= 24
\end{aligned}
$$

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1$ $v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2$ $v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5$ $v_3 = 18$	0	1	6	7	7	18	19	24	25	25	25	25
$w_4 = 6$ $v_4 = 22$	0	1	6	7	7	18	22	24	28	29	29	40
$w_5 = 7$ $v_5 = 28$	0											

Finally, the output is:

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1$ $v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2$ $v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5$ $v_3 = 18$	0	1	6	7	7	18	19	24	25	25	25	25
$w_4 = 6$ $v_4 = 22$	0	1	6	7	7	18	22	24	28	29	29	40
$w_5 = 7$ $v_5 = 28$	0	1	6	7	7	18	22	28	29	34	35	40

The maximum value of items in the knapsack is 40, the bottom-right entry). The dynamic programming approach can now be coded as the following algorithm

5. Pseudo code

```
Dynamic-0-1-knapsack (v, w, n, W)
for w = 0 to W do
  c[0, w] = 0
for i = 1 to n do
  c[i, 0] = 0
  for w = 1 to W do
    if wi ≤ w then
      if vi + c[i-1, w-wi] then
        c[i, w] = vi + c[i-1, w-wi]
      else c[i, w] = c[i-1, w]
    else
      c[i, w] = c[i-1, w]
```

6. Java Code

```
// for 0-1 Knapsack problem
class Knapsack {


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

	/
	static int knapSack(int W, int wt[],int val[], int n)
	{
		int i, w;
		int K[][] = new int[n + 1][W + 1];
		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];
	}

	// Driver code
	public static void main(String args[])
	{
		int val[] = new int[] { 60, 100, 120 };
		int wt[] = new int[] { 10, 20, 30 };
		int W = 50;
		int n = val.length;
		System.out.println(knapSack(W, wt, val, n));
	}
}
```

7. Simulation Screen shots.

source code

```
{

    // Declare the table dynamically
    int dp[][] = new int[N + 1][W + 1];

    // Loop to initially filled the
    // table with -1
    for(int i = 0; i < N + 1; i++)
        for(int j = 0; j < W + 1; j++)
            dp[i][j] = -1;

    return knapSackRec(W, wt, val, N, dp);
}

// Driver Code
public static void main(String [] args)
{
    int val[] = { 60, 100, 120 };
    int wt[] = { 10, 20, 30 };

    int W = 50;
    int N = val.length;

    System.out.println(knapSack(W, wt, val, N));
}
}

// This Code is contributed By FARAZ AHMAD
```

input

```
220


...Program finished with exit code 0
Press ENTER to exit console.
```

8. Time Complexity & Space Complexity.

Time Complexity: O(N*W).
As redundant calculations of states are avoided.
Auxiliary Space: O(N*W).
The use of 2D array data structure for storing intermediate states