72017912C
COMPBEB1326

GROUP A : ASSIGNMENT No. 4

Page No.
Date

Title: 0-1 Knapsack Problem

Objective: To solve 0-1 Knapsack Problem using dynamic programming or branch & bound strategy.

Problem Statement:

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

Software & Hardware Requirement:

1. Desktop / Laptop
2. Any Operating System
3. Python
4. IDE or Code Editor

Theory:

Knapsack Problem:

You are given -

1) A knapsack with limited weight capacity
2) Few items each having some weight & value.

Items should be placed into the knapsack such that -

1) The value of profit obtained by putting the items into the knapsack is maximum.
2) And the weight limit of knapsack does not exceed.

0/1 Knapsack Problem:

In 0/1 Knapsack Problem,

1) As the name suggests items are indivisible here.

2) We can not take fraction of any item.
3) We have to either take an item completely or leave it completely.
4) It is solved using dynamic programming approach

Dynamic Programming:

Dynamic Programming is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming. The idea is to simply store the results of subproblems, so that we do not have to re-compute them when needed later. This simple optimization reduces time complexities from exponential to polynomial.

Problem:

~~A thief is robbing a store & can carry a maximal weight of W into his knapsack. There are n items & weight of i^{th} item is w_i & the profit of selecting this item is p_i. What item thief should take?~~

Dynamic Programmig Approach:

Let i be the highest numbered item in an optimal solution S for W dollars. Then $S' = S - \{i\}$ is an optimal solution for $W - w_i$ dollars & the value of the solution S is V_i plus the value of the subproblem.

We can express this fact in the following formula. define $[i, w]$ to be the solution for items $1, 2, \ldots, i$ & the maximum weight w.

Page No.			
Date			

The algorithm takes the following inputs
The maximum weight W
The number of items n
The two sequences $v = \langle v_1, v_2, \dots v_n \rangle$ & $w = \langle w_1, w_2, \dots w_n \rangle$

Algorithm:

```
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]
```

The set of items to take can be deduced from the table, starting at $C[n, W]$ & tracing backwords where the optimal values came from.

If $C[i, W] = C[i-1, w]$, then item i is not part of the solution, & we continue tracing with $C[i-1, w]$. Otherwise, item i is part of the solution, & we continue tracing with $C[i-1, w-W]$.

Page No.
Date

Analysis:

This algorithm takes $\Theta(n, W)$ times as table C has $(n+1)(W+1)$ entries, where each entry requires $\Theta(1)$ time to compute.

Test Cases:

Input:

Item	A	B	C	D
Profit	24	18	18	10
Weight	24	10	10	7

Expected Output = 36

Actual Output = 18+18 = 36

Result: Pass

Conclusion:

Successfully implemented 0-1 Knapsack problem using Dynamic Programming.

Bumbre 28/9/22

Code:

```
def knapSack(W,wt,val,n):

    dp = [0 for i in range(W+1)]

    for i in range(0,n):
        for w in range(W,0,-1):
            if wt[i] <= w:
                dp[w] = max(dp[w], dp[w-wt[i]]+val[i])

    return dp[W]

val = [2,3,5,1,3]
wt = [1,2,3,4,5]

W = 10
n = len(val)

print(knapSack(W,wt,val,n))
```

Output:

```
11
```