**GROUP – A**

**Assignment No: 4**

====================================================================

Title:-

Write a program to solve 0-1 knapsack problem using dynamic programming

=====================================================================Objective:-

-To learn about 0-1 knapsack

-To understand dynamic method

=====================================================================

Theory:-

In dynamic programming method, the solving a complex problem by first breaking into a collection of simpler subproblems. Solving each subproblem just once and then storing their solutions to avoid repetitive computations.

* What is 0-1 knapsack problem?

In knapsack problem, the knapsack is like container or bag. Some items which have some weights or profit, it has to put into the knapsack in such way that the total value produces maximum profit.

In 0-1 knapsack problem, the items are either completely or no item are filled in knapsack.

* How 0-1 knapsack problem works?

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. It cannot break an item, either pick the complete item or don’t pick it (0-1 property).

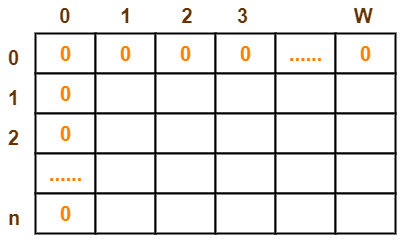
* 0-1 knapsack using dynamic method

Consider,

Knapsack weight capacity = w

Number of items each having some weight and value = n

* Step-1: Draw a table say ‘T’ with (n+1) number of rows and (w+1) number of columns. Fill all the boxes of 0th row and 0th column with zeroes as shown -



* Step-2: Start filling the table row wise top to bottom from left to right. Use the following formula-

**T (i , j) = max { T ( i-1 , j ) , valuei + T( i-1 , j – weighti) }**

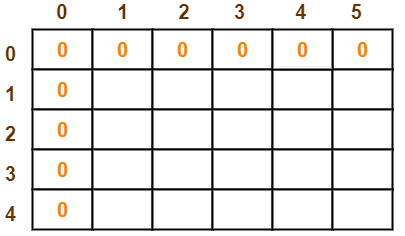
Here, T(i , j) = maximum value of the selected items if we can take items 1 to i and have weight restrictions of j.

This step leads to completely filling the table. Then, value of the last box represents the maximum possible value that can be put into the knapsack.

* Step-3: To identify the items that must be put into the knapsack to obtain that maximum profit,
* Consider the last column of the table.
* Start scanning the entries from bottom to top.
* On encountering an entry whose value is not same as the value stored in the entry immediately above it, mark the row label of that entry.
* After all the entries are scanned, the marked labels represent the items that must be put into the knapsack.
* Practice problem based on 0-1 knapsack using dynamic programming method-
* Problem : -For the given set of items and knapsack capacity = 5 kg, find the optimal solution for the 0/1 knapsack problem making use of dynamic programming approach.

|  |  |  |
| --- | --- | --- |
| **Item** | **Weight** | **Value** |
| 1 | 2 | 3 |
| 2 | 3 | 4 |
| 3 | 4 | 5 |
| 4 | 5 | 6 |

* Step 1: Draw a table say ‘T’ with (n+1) = 4 + 1 = 5 number of rows and (w+1) = 5 + 1 = 6 number of columns. Fill all the boxes of 0th row and 0th column with 0.



* Step 2: Start filling the table row wise top to bottom from left to right using the formula-

**T (i , j) = max { T ( i-1 , j ) , valuei + T( i-1 , j – weighti) }**

**Finding T(1,1)-**

 i = 1

j = 1

(value)i = (value)1 = 3

(weight)i = (weight)1 = 2

Substituting the values-

T(1,1) = max { T(1-1 , 1) , 3 + T(1-1 , 1-2) }

T(1,1) = max { T(0,1) , 3 + T(0,-1) }

T(1,1) = T(0,1) { Ignore T(0,-1) }

T(1,1) = 0

**Finding T(1,2)-**

 i = 1

j = 2

(value)i = (value)1 = 3

(weight)i = (weight)1 = 2

Substituting the values, we get-

T(1,2) = max { T(1-1 , 2) , 3 + T(1-1 , 2-2) }

T(1,2) = max { T(0,2) , 3 + T(0,0) }

T(1,2) = max {0 , 3+0}

T(1,2) = 3

**Finding T(1,3)-**

 i = 1

j = 3

(value)i = (value)1 = 3

(weight)i = (weight)1 = 2

Substituting the values-

T(1,3) = max { T(1-1 , 3) , 3 + T(1-1 , 3-2) }

T(1,3) = max { T(0,3) , 3 + T(0,1) }

T(1,3) = max {0 , 3+0}

T(1,3) = 3

**Finding T(1,4)-**

 i = 1

j = 4

(value)i = (value)1 = 3

(weight)i = (weight)1 = 2

Substituting the values-

T(1,4) = max { T(1-1 , 4) , 3 + T(1-1 , 4-2) }

T(1,4) = max { T(0,4) , 3 + T(0,2) }

T(1,4) = max {0 , 3+0}

T(1,4) = 3

**Finding T(1,5)-**

 i = 1

j = 5

(value)i = (value)1 = 3

(weight)i = (weight)1 = 2

Substituting the values-

T(1,5) = max { T(1-1 , 5) , 3 + T(1-1 , 5-2) }

T(1,5) = max { T(0,5) , 3 + T(0,3) }

T(1,5) = max {0 , 3+0}

T(1,5) = 3

**Finding T(2,1)-**

 i = 2

j = 1

(value)i = (value)2 = 4

(weight)i = (weight)2 = 3

Substituting the values-

T(2,1) = max { T(2-1 , 1) , 4 + T(2-1 , 1-3) }

T(2,1) = max { T(1,1) , 4 + T(1,-2) }

T(2,1) = T(1,1) { Ignore T(1,-2) }

T(2,1) = 0

**Finding T(2,2)-**

 i = 2

j = 2

(value)i = (value)2 = 4

(weight)i = (weight)2 = 3

Substituting the values-

T(2,2) = max { T(2-1 , 2) , 4 + T(2-1 , 2-3) }

T(2,2) = max { T(1,2) , 4 + T(1,-1) }

T(2,2) = T(1,2) { Ignore T(1,-1) }

T(2,2) = 3

**Finding T(2,3)-**

i = 2

j = 3

(value)i = (value)2 = 4

(weight)i = (weight)2 = 3

Substituting the values-

T(2,3) = max { T(2-1 , 3) , 4 + T(2-1 , 3-3) }

T(2,3) = max { T(1,3) , 4 + T(1,0) }

T(2,3) = max { 3 , 4+0 }

T(2,3) = 4

**Finding T(2,4)-**

 i = 2

j = 4

(value)i = (value)2 = 4

(weight)i = (weight)2 = 3

Substituting the values-

T(2,4) = max { T(2-1 , 4) , 4 + T(2-1 , 4-3) }

T(2,4) = max { T(1,4) , 4 + T(1,1) }

T(2,4) = max { 3 , 4+0 }

T(2,4) = 4

**Finding T(2,5)-**

i = 2

j = 5

(value)i = (value)2 = 4

(weight)i = (weight)2 = 3

Substituting the values-

T(2,5) = max { T(2-1 , 5) , 4 + T(2-1 , 5-3) }

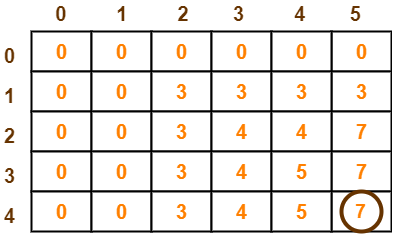
T(2,5) = max { T(1,5) , 4 + T(1,2) }

T(2,5) = max { 3 , 4+3 }

T(2,5) = 7

Similarly, compute all the entries.

After all the entries are computed and filled in the table, we get the following table-



* Step 3: Identify items and put it in knapsack

The last entry represents the maximum possible value that can be put into the knapsack.

So, maximum possible value that can be put into the knapsack = 7.

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]

* Time Complexity and Space Complexity of 0-1 knapsack using dynamic programming

Each entry of the table requires constant time θ(1) for its computation. It takes θ(nw) time to fill (n+1)(w+1) table entries. It takes θ(n) time for tracing the solution since tracing process traces the n rows. Thus, overall θ(nw) time is taken to solve 0/1 knapsack problem using dynamic programming.

=====================================================================

Conclusion:-

Thus we have studied to how 0-1 knapsack using dynamic programming works. The use of dynamic programming method is Solving each subproblem just once and then storing their solutions to avoid repetitive computations.

=====================================================================