Design and Analysis of Algorithms

- ***Course Code***: BCSE304L
- ***Course Type***: Theory (ETH)
- ***Slot***: A1+TA1 & & A2+TA2
- ***Class ID***: VL2023240500901
  VL2023240500902

A1+TA1

Day	Start	End
Monday	08:00	08:50
Wednesday	09:00	09:50
Friday	10:00	10:50

A2+TA2

Day	Start	End
Monday	14:00	14:50
Wednesday	15:00	15:50
Friday	16:00	16:50

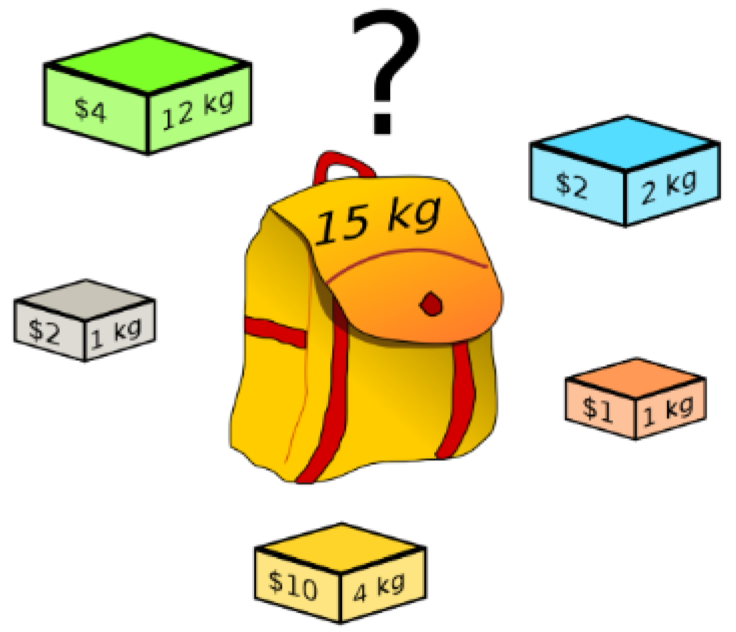
Syllabus- Module 2

Module:2	Design Paradigms: Dynamic Programming, Backtracking and Branch & Bound Techniques	10 hours

Dynamic programming: Assembly Line Scheduling, Matrix Chain Multiplication, Longest Common Subsequence, 0-1 Knapsack, TSP
Backtracking: N-Queens problem, Subset Sum, Graph Coloring
Branch & Bound: LIFO-BB and FIFO BB methods: Job Selection problem, 0-1 Knapsack Problem

0/1 Knapsack Problem Using Dynamic Programming

Knapsack Problem (introduction)



Knapsack Problem

- The basic scenario of the Knapsack Problem involves a knapsack (or backpack) with a limited carrying capacity and a set of items, each with a **weight** and a **value**.

The objective is to determine the combination of items to include in the knapsack in such a way that the total value is maximized, while the total weight does not exceed the capacity of the knapsack.

To reiterate:

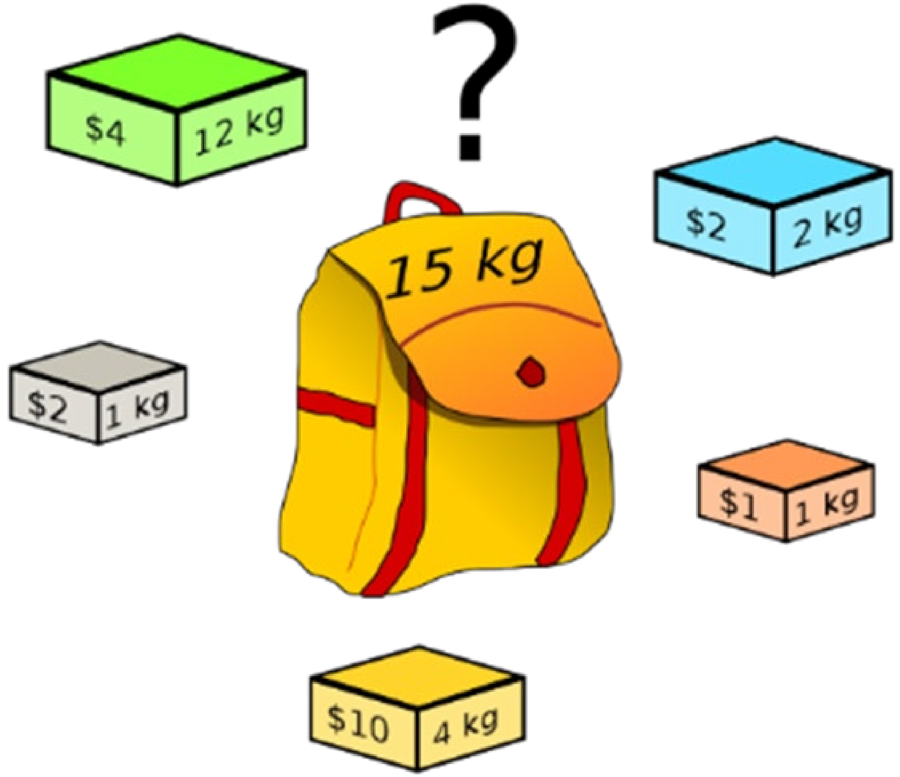
- **Objective**: Maximize the total value or profit of the items selected for the knapsack.
- **Constraint**: The total weight of the selected items must not exceed the weight limit (capacity) of the knapsack.

0/1 Knapsack Problem Using Dynamic Programming

Input: A set of items, each with a weight w_i and a value v_i, where i is the index of the item.

The maximum capacity of the knapsack, denoted as W .

Output: A binary vector x of length n, where x_i is 1 if item i is selected and 0 otherwise.



Objective: Maximize the total value $\sum_{i=1}^{n} x_i \times v_i$

Subject to: The total weight of selected items must not exceed the knapsack capacity: $\sum_{i=1}^{n} x_i \times w_i \leq W$

0/1 Knapsack Problem Using Dynamic Programming

Sample Input and Output:

Knapsack Capacity (W) = 5

Three Items:

- I1: Weight: 2, Value: 3
- I2: Weight: 1, Value: 2
- I3: Weight: 3, Value: 4

The output should provide the maximum value that can be obtained within the given capacity.

In this case, the output might be 7 (by selecting items 1 and 3).

Three Items:

- I1: Weight: 2, Value: 3
- I2: Weight: 1, Value: 2
- I3: Weight: 3, Value: 4

0/1 Knapsack Problem Using Dynamic Programming

Time Complexity in Brute Force Approach:
The brute force approach has an exponential time complexity of O(2^n) since it considers all possible subsets of items

How?
This is because for each item, you have two options (include or exclude), and when you have 'n' items, the total number of possible combinations is 2^n.

Explanation:

- **For the first item**: You have 2 choices (include or exclude).
- **For the second item**: For each choice from the first item, you have 2 more choices, resulting in 2 * 2 = 2^2 = 4 combinations.
- **For the third item**: For each choice from the second item, you have 2 more choices, resulting in 2 * 2 * 2 = 2^3 = 8 combinations.
- This pattern continues, and in general, for **'n' items**, you have **2^n** possible combinations.

0/1 Knapsack Problem Using Dynamic Programming (Procedure)

- Initialize a table, often called dp, of size **(n+1) x (W+1)**, where n is the number of items and W is the knapsack capacity.
- **Base Case**: Fill in the table for the base cases, i.e., when the number of items is 0 or the knapsack capacity is 0.
- **Fill the Table**: Iterate over each item and knapsack capacity, and fill in the table based on the recurrence relation.
- **Retrieve Solution**: Once the table is filled, the final result is usually found in dp[n][W], where n is the number of items and W is the knapsack capacity.

Recurrence relation

$$\boxed{dp[i][w]=\max(dp[i-1][w],dp[i-1][w-weight[i]]+value[i])}$$

dp[i][w] represents the maximum value achievable considering the first i items and a knapsack capacity of w.
The relation considers either excluding the i-th item or including it in the knapsack, choosing the option that yields a higher value.

0/1 Knapsack Problem Using Dynamic Programming (Procedure)

Recurrence relation

$$\boxed{dp[i][w]=\max(dp[i-1][w],dp[i-1][w-weight[i]]+value[i])}$$

dp[i][w] represents the maximum value achievable considering the first i items and a knapsack capacity of w.

The relation considers either excluding the i-th item or including it in the knapsack, choosing the option that yields a higher value.

Time Complexity in Dynamic Programming:

The dynamic programming approach significantly improves the time complexity to $O(nW)$ where

- n is the number of items.
- W is the knapsack capacity.

0/1 Knapsack Problem Using Dynamic Programming

Knapsack weight capacity = W; Number of items each having some weight and value = n

0/1 knapsack problem is solved using dynamic programming in the following steps-

Step-01: Draw the Table

Create a table 'T' with (n+1) rows and (W+1) columns.

The rows correspond to the items (0 to n), and the columns represent the weights (0 to w).

Initialize the first row and first column with zeros since they represent the base cases.

	0	1	2	3		W
0	0	0	0	0	......	0
1	0					
2	0					
	......					
n	0					

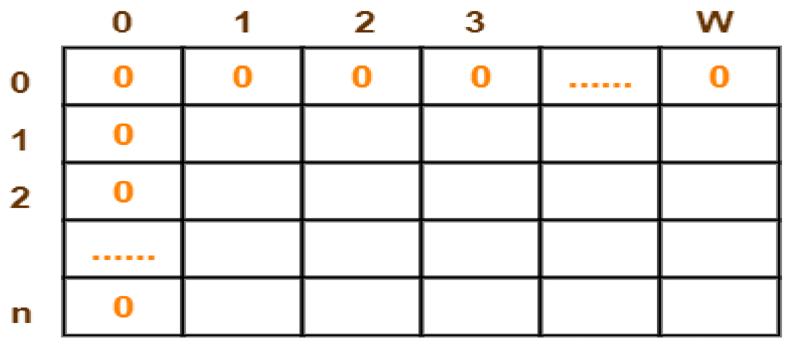
T-Table

0/1 Knapsack Problem Using Dynamic Programming

Knapsack weight capacity = w; Number of items each having some weight and value = n

0/1 knapsack problem is solved using dynamic programming in the following steps-

Step-01: Draw the Table
Create a table 'T' with (n+1) rows and (w+1) columns.
The rows correspond to the items (0 to n), and the columns represent the weights (0 to w).
Initialize the first row and first column with zeros since they represent the base cases.



	0	1	2	3		W
0	0	0	0	0	......	0
1	0					
2	0					
	......					
n	0					

T-Table

Step-02: Fill the Table

- Iterate through each row (item) and column (weight) from top to bottom and left to right.
- Use the recurrence relation to fill in the table:

$$T(i, j) = \max\{T(i-1, j), value_i + T(i-1, j - weighti)\}$$

Here, $T(i, j)$ represents the maximum value achievable considering the first i items and a knapsack capacity of j.

0/1 Knapsack Problem Using Dynamic Programming

Knapsack weight capacity = w; Number of items each having some weight and value = n

0/1 knapsack problem is solved using dynamic programming in the following steps-

Step-01: Draw the Table
Create a table 'T' with (n+1) rows and (w+1) columns.
The rows correspond to the items (0 to n), and the columns represent the weights (0 to w).
Initialize the first row and first column with zeros since they represent the base cases.

Step-02: Fill the Table

- Iterate through each row (item) and column (weight) from top to bottom and left to right.
- Use the recurrence relation to fill in the table:

$$T(i, j) = \max\{T(i-1, j), value_i + T(i-1, j - weighti)\}$$

Here, $T(i, j)$ represents the maximum value achievable considering the first i items and a knapsack capacity of j.

Step-03: Identify the Items

- To identify the items that must be put into the knapsack to obtain the maximum profit, focus on the last column of the table (representing the maximum weight capacity).
- Scan the entries from bottom to top.
  - When you encounter an entry whose value is different from the value stored in the entry immediately above it, mark the row label of that entry.
- After scanning all entries, the marked row labels represent the items that must be included in the knapsack to achieve the maximum profit.

0/1 Knapsack Problem Using Dynamic Programming: Example

n = 4, w = 5 kg; (w1, w2, w3, w4) = (2, 3, 4, 5); (b1, b2, b3, b4) = (3, 4, 5, 6)

Step-01: 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.

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

T-Table

Step-02: Start filling the table row wise top to bottom from left to right using the formula-

$$\mathbf{T(i , j) = max\{T (i-1 , j) , value_i + T(i-1 , j - weight_i) \}}$$

Finding T(1,1)-

We have,
i = 1
j = 1
$(value)_i = (value)_1 = 3$
$(weight)_i = (weight)_1 = 2$

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

0/1 Knapsack Problem Using Dynamic Programming: Example

n = 4, w = 5 kg; (w1, w2, w3, w4) = (2, 3, 4, 5); (b1, b2, b3, b4) = (3, 4, 5, 6)

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

T-Table

Step-02: **$T(i, j) = \max\{T(i-1, j), \text{value}_i + T(i-1, j - \text{weight}_i)\}$**

Finding T(1,1)-

i = 1

j = 1

$(\text{value})_i = (\text{value})_1 = 3$

$(\text{weight})_i = (\text{weight})_1 = 2$

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

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

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

$T(1,1) = 0$

T(1,2)-

i = 1

j = 2

$(\text{value})_i = (\text{value})_1 = 3$

$(\text{weight})_i = (\text{weight})_1 = 2$

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

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

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

$T(1,2) = 3$

T(1,3)-

i = 1

j = 3

$(\text{value})_i = (\text{value})_1 = 3$

$(\text{weight})_i = (\text{weight})_1 = 2$

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

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

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

$T(1,3) = 3$

T(1,4)-

i = 1

j = 4

$(\text{value})_i = (\text{value})_1 = 3$

$(\text{weight})_i = (\text{weight})_1 = 2$

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

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

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

$T(1,4) = 3$

T(1,5)-

i = 1

j = 5

$(\text{value})_i = (\text{value})_1 = 3$

$(\text{weight})_i = (\text{weight})_1 = 2$

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

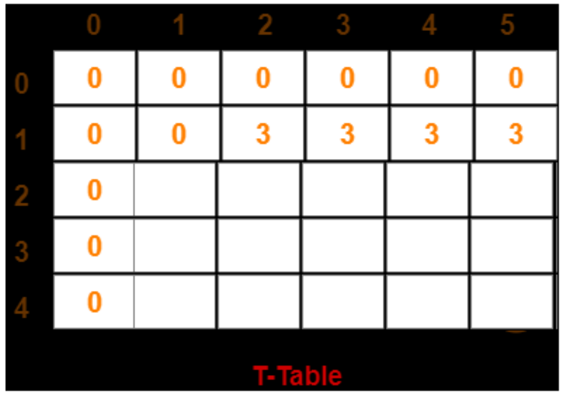
$T(1,5) = \max\{T(0,5), 3 + T(0,3)\}$

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

$T(1,5) = 3$

0/1 Knapsack Problem Using Dynamic Programming: Example

n = 4, w = 5 kg; (w1, w2, w3, w4) = (2, 3, 4, 5); (b1, b2, b3, b4) = (3, 4, 5, 6)



	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0					
3	0					
4	0					

T-Table

Step-02: **T (i , j) = max{T (i-1 , j) , value$_i$ + T(i-1 , j – weight$_i$) }**

Finding T(1,1)-

i = 1

j = 1

(value)$_i$ = (value)$_1$ = 3

(weight)$_i$ = (weight)$_1$ = 2

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

T(1,2)-

i = 1

j = 2

(value)$_i$ = (value)$_1$ = 3

(weight)$_i$ = (weight)$_1$ = 2

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

T(1,3)-

i = 1

j = 3

(value)$_i$ = (value)$_1$ = 3

(weight)$_i$ = (weight)$_1$ = 2

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

T(1,4)-

i = 1

j = 4

(value)$_i$ = (value)$_1$ = 3

(weight)$_i$ = (weight)$_1$ = 2

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

T(1,5)-

i = 1

j = 5

(value)$_i$ = (value)$_1$ = 3

(weight)$_i$ = (weight)$_1$ = 2

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

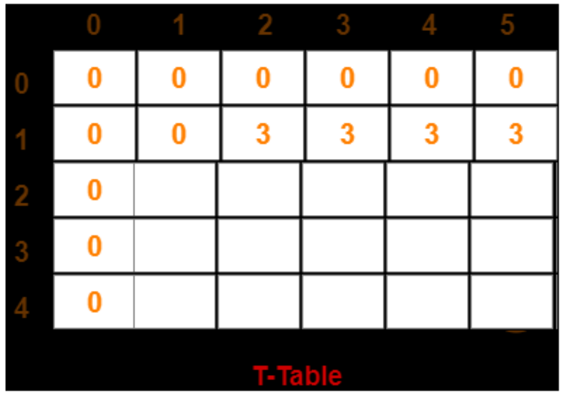
T(1,5) = max{T(0,5) , 3 + T(0,3) }

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

T(1,5) = 3

0/1 Knapsack Problem Using Dynamic Programming: Example

n = 4, w = 5 kg; (w1, w2, w3, w4) = (2, 3, 4, 5); (b1, b2, b3, b4) = (3, 4, 5, 6)



	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0					
3	0					
4	0					

T-Table

Step-02: T (i , j) = max{T (i-1 , j) , value$_i$ + T(i-1 , j – weight$_i$) }

T(2,1)
i = 2
j = 1
(value)i = (value)2 = 4
(weight)i = (weight)2 = 3
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

T(2,2)
i = 2
j = 2
(value)i = (value)2 = 4
(weight)i = (weight)2 = 3
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

T(2,3)
i = 2
j = 3
(value)i = (value)2 = 4
(weight)i = (weight)2 = 3
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

T(2,4)
i = 2
j = 4
(value)i = (value)2 = 4
(weight)i = (weight)2 = 3
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

T(2,5)
i = 2
j = 5
(value)i = (value)2 = 4
(weight)i = (weight)2 = 3
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

0/1 Knapsack Problem Using Dynamic Programming: Example

$n = 4, \quad w = 5 \text{ kg}$

$(\text{w1, w2, w3, w4}) = (2, 3, 4, 5)$

$(\text{b1, b2, b3, b4}) = (3, 4, 5, 6)$

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

T-Table

Filling T(1, j) for j = 1 to 5:

- $T(1,1) = 0$ (no space for item-1 in a knapsack of weight 1)
- $T(1,2) = 3$ (value of item-1 can fit in knapsack of weight 2)
- $T(1,3) = 3$
- $T(1,4) = 3$
- $T(1,5) = 3$

Filling T(2, j) for j = 1 to 5:

- $T(2,1) = 0$ (no space for item-2 in a knapsack of weight 1)
- $T(2,2) = 3$ (value of item-1 can fit, ignore item-2)
- $T(2,3) = 4$ (value of item-2 can fit in knapsack of weight 3)
- $T(2,4) = 4$
- $T(2,5) = 7$ (value of item-2 and item-1 can fit)

Filling T(3, j) for j = 1 to 5:

- $T(3,1) = 0$
- $T(3,2) = 3$
- $T(3,3) = 4$
- $T(3,4) = 5$ (value of item-3 can fit in knapsack of weight 4)
- $T(3,5) = 7$

Filling T(4, j) for j = 1 to 5:

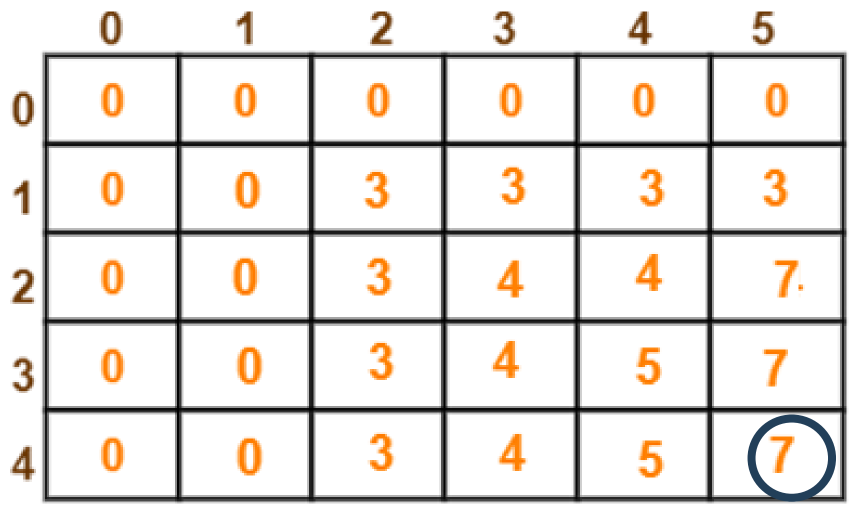
- $T(4,1) = 0$
- $T(4,2) = 3$
- $T(4,3) = 4$
- $T(4,4) = 5$
- $T(4,5) = 7$

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7.
3	0	0	3	4	5	7
4	0	0	3	4	5	7

T-Table

n = 4, w = 5 kg; (w1, w2, w3, w4) = (2, 3, 4, 5); (b1, b2, b3, b4) = (3, 4, 5, 6)

- 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.



	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

T-Table

Identifying Items To Be Put Into Knapsack

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

T-Table

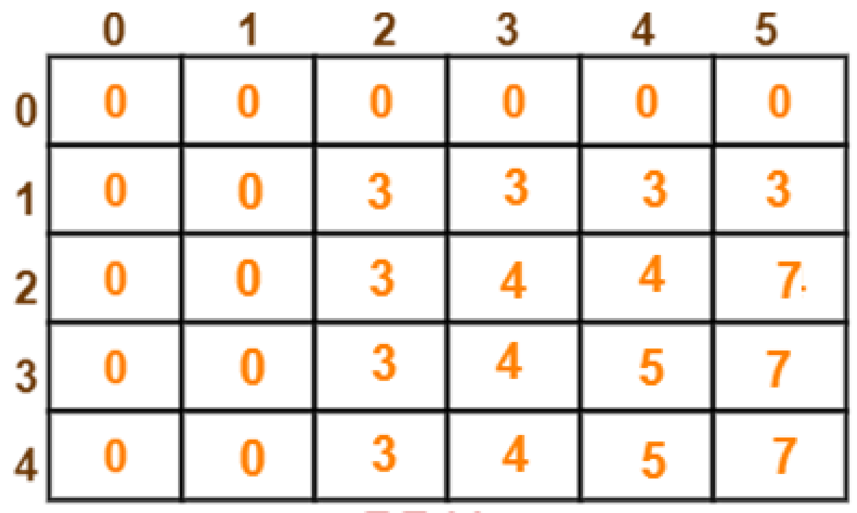
- Start scanning the entries from bottom to top.
- On encountering an entry whose value is not same as the value stored in the entry immediately above it, mark the row label of that entry.
- After all the entries are scanned, the marked labels represent the items that must be put into the knapsack.

•We mark the rows labelled "1" and "2".
•Thus, items that must be put into the knapsack to obtain the maximum value 7 are-
Item-1 and Item-2

	0	1	2	3	4	5
0	0	0	0	0	0	0
✓ 1	0	0	3	3	3	3
✓ 2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

T-Table

```
**Algorithm** DP(){
Input: $\{w_1, w_2, w_3, \ldots, w_n\}$, C, $\{p_1, p_2, p_3, \ldots, p_n\}$
Output: T[n+1, C+1]
For i=0 to C+1 {
        T[0, i]=0
}
For i=0 to n+1 {
T[i, 0]=0
For(j=1 to C+1) {
If($w_i$<=j) and T[i-1, j-$w_i$] + $p_i$ >T[i-1, j]
Then T[i, j] = T[i-1, j-$w_i$] + $p_i$
Else T[i, j]=T[i-1, j]
}
```



	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

T-Table

Step-02: Start filling the table row wise top to bottom from left to right using the formula: **T (i , j) = max{T (i-1 , j) , value$_i$ + T(i-1 , j – weight$_i$) }**

Important Analysis

- Each entry of the table requires constant time O(1) for its computation.
- It takes O(nw) time to fill (n+1)(w+1) table entries.

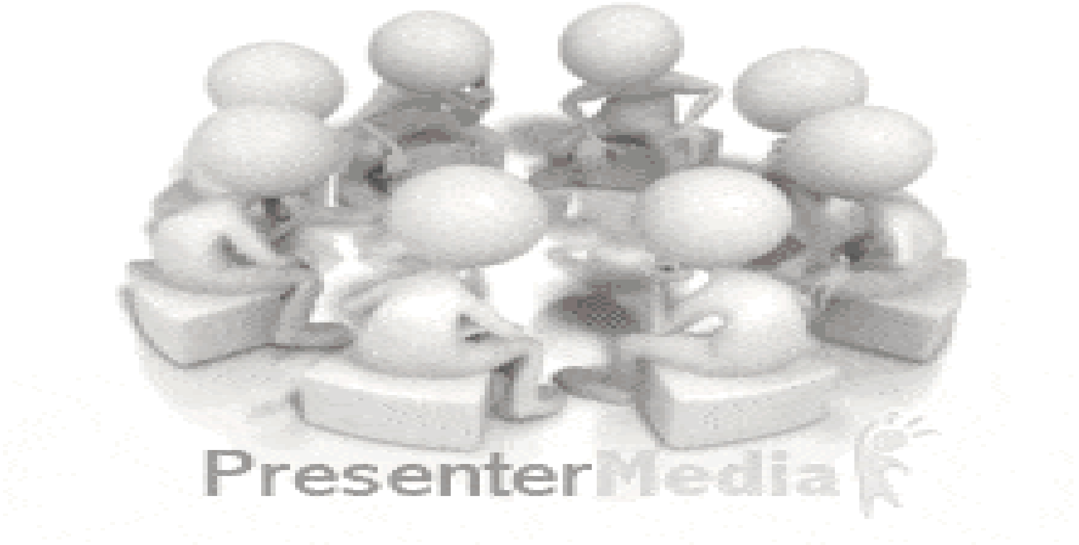
We know the time complexity of 0/1 knapsack problem in brute force approach is $O(2^n)$

Time complexity of 0/1 knapsack problem in Dynamic programming is O(NW).

Which one is preferable?

- That depends on the value of W.
  - If the value of W is very less, go for Dynamic programming.
  - If the value of W is very high, compared to 2^n then go for normal method i.e., brute force approach.

Any
Question


PresenterMedia

Thank You!

FOR YOUR
ATTENTION