[0/1 Knapsack Problem | Dynamic Programming | Example](#)
[Design & Analysis of Algorithms](#)

Knapsack Problem-

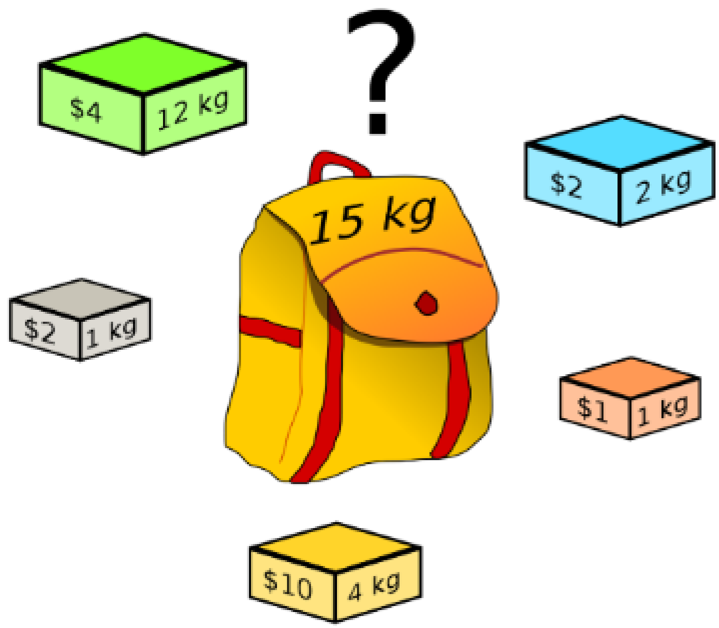
You are given the following-

- A knapsack (kind of shoulder bag) with limited weight capacity.
- Few items each having some weight and value.

The problem states-

Which items should be placed into the knapsack such that-

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



Knapsack Problem

Knapsack Problem Variants-

Knapsack problem has the following two variants-

1. Fractional Knapsack Problem
2. 0/1 Knapsack Problem

In this article, we will discuss about 0/1 Knapsack Problem.

0/1 Knapsack Problem-

In 0/1 Knapsack Problem,

- As the name suggests, items are indivisible here.
- We can not take the fraction of any item.
- We have to either take an item completely or leave it completely.
- It is solved using dynamic programming approach.

0/1 Knapsack Problem Using Dynamic Programming-

Consider-

- 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 a table say 'T' with (n+1) number of rows and (w+1) number of columns.
- Fill all the boxes of 0^{th} row and 0^{th} column with zeroes as shown-

	0	1	2	3		W
0	0	0	0	0	……	0
1	0					
2	0					
	……					
n	0					

T-Table

Step-02:

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

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

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-03:

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.

Time Complexity-

- 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.

PRACTICE PROBLEM BASED ON 0/1 KNAPSACK PROBLEM-

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

OR

Find the optimal solution for the 0/1 knapsack problem making use of dynamic programming approach. Consider-

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

OR

A thief enters a house for robbing it. He can carry a maximal weight of 5 kg into his bag. There are 4 items in the house with the following weights and values. What items should thief take if he either takes the item completely or leaves it completely?

Item	Weight (kg)	Value ($)
Mirror	2	3
Silver nugget	3	4
Painting	4	5
Vase	5	6

Solution-

Given-

- Knapsack capacity (w) = 5 kg
- Number of items (n) = 4

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 0^{th} row and 0^{th} 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-

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

Finding T(1,1)-

We have,

- $i = 1$
- $j = 1$
- $(value)_i = (value)_1 = 3$
- $(weight)_i = (weight)_1 = 2$

Substituting the values, we get-

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)-

We have,

- $i = 1$
- $j = 2$
- $(value)_i = (value)_1 = 3$
- $(weight)_i = (weight)_1 = 2$

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)-

We have,

- $i = 1$
- $j = 3$
- $(value)_i = (value)_1 = 3$
- $(weight)_i = (weight)_1 = 2$

Substituting the values, we get-

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)-

We have,

- $i = 1$
- $j = 4$
- $(value)_i = (value)_1 = 3$
- $(weight)_i = (weight)_1 = 2$

Substituting the values, we get-

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)-

We have,

- $i = 1$
- $j = 5$
- $(value)_i = (value)_1 = 3$
- $(weight)_i = (weight)_1 = 2$

Substituting the values, we get-

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)-

We have,

- $i = 2$
- $j = 1$
- $(value)_i = (value)_2 = 4$
- $(weight)_i = (weight)_2 = 3$

Substituting the values, we get-

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)-

We have,

- i = 2
- j = 2
- $(value)_i = (value)_2 = 4$
- $(weight)_i = (weight)_2 = 3$

Substituting the values, we get-
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)-

We have,

- i = 2
- j = 3
- $(value)_i = (value)_2 = 4$
- $(weight)_i = (weight)_2 = 3$

Substituting the values, we get-
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)-

We have,

- i = 2
- j = 4
- $(value)_i = (value)_2 = 4$
- $(weight)_i = (weight)_2 = 3$

Substituting the values, we get-
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)-

We have,

- i = 2
- j = 5
- $(value)_i = (value)_2 = 4$

- $(\text{weight})_i = (\text{weight})_2 = 3$

Substituting the values, we get-
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-

	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

- 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.

Identifying Items To Be Put Into Knapsack-

Following Step-04,
- 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

PROGRAM

```
public class KSP {
 // maximum of two integers
   static int max(int a, int b)   { return (a > b) ? a : b; }
// main
   public static void main(String args[])
   {
      // input  // هذا مثال معطيات يمكن تغييرها
      int n = 5;
      int val[] = new int[] { 60, 100, 120, 50 , 250 };
      int wt[] = new int[] { 10, 20, 30, 41,65 };
      int sumwt = 0;
      for(int i=0; i<n;i++) sumwt+= wt[i];
      double ratio =0.5;
```

```
      int W = (int) ((int) sumwt*ratio);
   // variables
      int i, w;
      int T[][] = new int[n + 1][W + 1];
   // Building table T[][] in bottom up manner
      for (i = 0; i<= n; i++) {
         for (w = 0; w<= W; w++) {
            if (i == 0 || w == 0)
               T[i][w] = 0;
            else if (wt[i - 1]<= w)
               T[i][w] = max(val[i - 1] + T[i - 1][w - wt[i - 1]], T[i - 1][w]);
            else
               T[i][w] = T[i - 1][w];
         }
      }
      System.out.println("The optimum of this instance is : " + T[n][W]);
// look for selected items
      System.out.println("Selected Items : " );
      while (n != 0) {
          if (T[n][W] != T[n - 1][W]) {
                  System.out.println("\tItem " + n + " with Weight = " + wt[n - 1] + " and Value = " + val[n - 1]);
                  W = W - wt[n-1];
          }
                          n--;
      }
   }
}
```