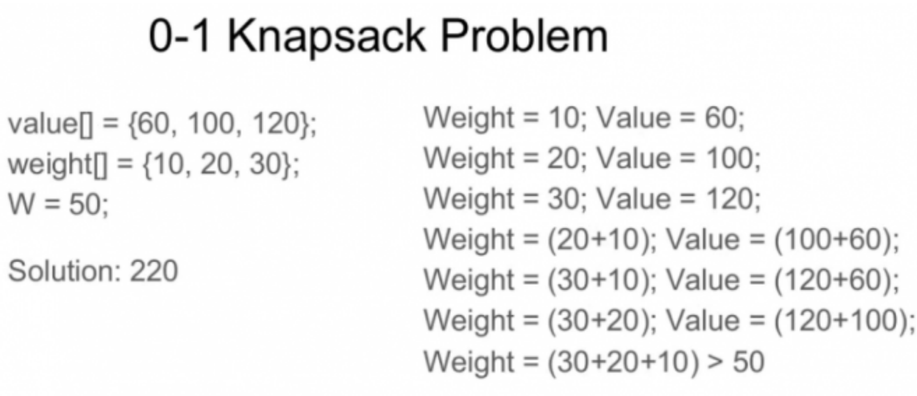
0-1 Knapsack Problem | DP-10

GeeksforGeeks
A computer science portal for geeks

Given weights and values of n ite pacity W to get the maximum total value in the knapsack. In other wo] 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).



Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.

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.

1) Optimal Substructure:

To consider all subsets of items, there can be two cases for every item: (1) the item is included in the optimal subset, (2) not included in the optimal set.

Therefore, the maximum value that can be obtained from n items is max of following two values.

1) Maximum value obtained by n-1 items and W weight (excluding nth item).

2) Value of nth item plus maximum value obtained by n-1 items and W minus weight of the nth item (including nth item).

If weight of nth item is greater than W, then the nth item cannot be included and case 1 is the only possibility.

2) Overlapping Subproblems

Following is recursive implementation that simply follows the recursive structure mentioned above.

C++

```
/* A Naive recursive implementation of 0-1 Knapsack problem */
#include <bits/stdc++.h>
using namespace std;

// A utility function that returns maximum of two integers
int max(int a, int b) { return (a > b)? a : b; }

// Returns the maximum value that
// can be put in a knapsack of capacity W
int knapSack(int W, int wt[], int val[], int n)
{

// Base Case
if (n == 0 || W == 0)
    return 0;

// If weight of the nth item is more
// than Knapsack capacity W, then
// this item cannot be included
// in the optimal solution
if (wt[n-1] > W)
    return knapSack(W, wt, val, n-1);

// Return the maximum of two cases:
// (1) nth item included
// (2) not included
else return max( val[n-1] + knapSack(W-wt[n-1], wt, val, n-1),
                    knapSack(W, wt, val, n-1) );
}

// Driver code
int main()
{
    int val[] = {60, 100, 120};
    int wt[] = {10, 20, 30};
    int W = 50;
    int n = sizeof(val)/sizeof(val[0]);
    cout<<knapSack(W, wt, val, n);
    return 0;
}

// This code is contributed by rathbhupendra
```

C

```
/* A Naive recursive implementation of 0-1 Knapsack problem */
#include<stdio.h>

// A utility function that returns maximum of two integers
int max(int a, int b) { return (a > b)? a : b; }
```

```
// Returns the maximum value that can be put in a knapsack of capacity W
int knapSack(int W, int wt[], int val[], int n)
{
   // Base Case
   if (n == 0 || W == 0)
       return 0;

   // If weight of the nth item is more than Knapsack capacity W, then
   // this item cannot be included in the optimal solution
   if (wt[n-1] > W)
       return knapSack(W, wt, val, n-1);

   // Return the maximum of two cases:
   // (1) nth item included
   // (2) not included
   else return max( val[n-1] + knapSack(W-wt[n-1], wt, val, n-1),
                    knapSack(W, wt, val, n-1)
                  );
}

// Driver program to test above function
int main()
{
    int val[] = {60, 100, 120};
    int wt[] = {10, 20, 30};
    int  W = 50;
    int n = sizeof(val)/sizeof(val[0]);
    printf("%d", knapSack(W, wt, val, n));
    return 0;
}
```

Java

```
/* A Naive recursive implementation of 0-1 Knapsack problem */
class Knapsack
{

    // A utility function that returns maximum of two integers
     static int max(int a, int b) { return (a > b)? a : b; }

     // Returns the maximum value that can be put in a knapsack of capacity W
     static int knapSack(int W, int wt[], int val[], int n)
     {
        // Base Case
    if (n == 0 || W == 0)
        return 0;

    // If weight of the nth item is more than Knapsack capacity W, then
    // this item cannot be included in the optimal solution
    if (wt[n-1] > W)
       return knapSack(W, wt, val, n-1);

    // Return the maximum of two cases:
    // (1) nth item included
    // (2) not included
    else return max( val[n-1] + knapSack(W-wt[n-1], wt, val, n-1),
                     knapSack(W, wt, val, n-1)
                      );
      }


   // Driver program to test above function
   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));
    }
}
/*This code is contributed by Rajat Mishra */
```

Python

```python
#A naive recursive implementation of 0-1 Knapsack Problem

# Returns the maximum value that can be put in a knapsack of
# capacity W
def knapSack(W , wt , val , n):

    # Base Case
    if n == 0 or W == 0 :
        return 0

    # If weight of the nth item is more than Knapsack of capacity
    # W, then this item cannot be included in the optimal solution
    if (wt[n-1] > W):
        return knapSack(W , wt , val , n-1)

    # return the maximum of two cases:
    # (1) nth item included
    # (2) not included
    else:
        return max(val[n-1] + knapSack(W-wt[n-1] , wt , val , n-1),
                   knapSack(W , wt , val , n-1))

# end of function knapSack

# To test above function
val = [60, 100, 120]
wt = [10, 20, 30]
W = 50
n = len(val)
print knapSack(W , wt , val , n)

# This code is contributed by Nikhil Kumar Singh
```

C#

```csharp
/* A Naive recursive implementation of
0-1 Knapsack problem */
using System;

class GFG {

    // A utility function that returns
    // maximum of two integers
    static int max(int a, int b)
    {
        return (a > b)? a : b;
    }

    // Returns the maximum value that can
    // be put in a knapsack of capacity W
    static int knapSack(int W, int []wt,
```

```
                        int []val, int n)
    {

        // Base Case
        if (n == 0 || W == 0)
            return 0;

        // If weight of the nth item is
        // more than Knapsack capacity W,
        // then this item cannot be
        // included in the optimal solution
        if (wt[n-1] > W)
            return knapSack(W, wt, val, n-1);

        // Return the maximum of two cases:
        // (1) nth item included
        // (2) not included
        else return max( val[n-1] +
            knapSack(W-wt[n-1], wt, val, n-1),
                   knapSack(W, wt, val, n-1));
    }

    // Driver function
    public static void Main()
    {
        int []val = new int[]{60, 100, 120};
        int []wt = new int[]{10, 20, 30};
        int W = 50;
        int n = val.Length;

        Console.WriteLine(knapSack(W, wt, val, n));
    }
}

// This code is contributed by Sam007
```

PHP

```
<?php
// A Naive recursive implementation
// of 0-1 Knapsack problem

// Returns the maximum value that
// can be put in a knapsack of
// capacity W
function knapSack($W, $wt, $val, $n)
{
    // Base Case
    if ($n == 0 || $W == 0)
        return 0;

    // If weight of the nth item is
    // more than Knapsack capacity
    // W, then this item cannot be
    // included in the optimal solution
    if ($wt[$n - 1] > $W)
        return knapSack($W, $wt, $val, $n - 1);

    // Return the maximum of two cases:
    // (1) nth item included
    // (2) not included
    else
        return max($val[$n - 1] +
              knapSack($W - $wt[$n - 1],
```

```
                  $wt, $val, $n - 1),
                  knapSack($W, $wt, $val, $n-1));
}

    // Driver Code
    $val = array(60, 100, 120);
    $wt = array(10, 20, 30);
    $W = 50;
    $n = count($val);
    echo knapSack($W, $wt, $val, $n);

// This code is contributed by Sam007
?>
```

Output:

```
220
```

It should be noted that the above function computes the same subproblems again and again. See the following recursion tree, K(1, 1) is being evaluated twice. Time complexity of this naive recursive solution is exponential (2^n).

```
In the following recursion tree, K() refers to knapSack().  The two
parameters indicated in the following recursion tree are n and W.
The recursion tree is for following sample inputs.
wt[] = {1, 1, 1}, W = 2, val[] = {10, 20, 30}

                       K(3, 2)         ---------> K(n, W)
                   /            \
                 /                \
            K(2,2)                  K(2,1)
          /       \                 /    \
        /           \             /        \
       K(1,2)      K(1,1)        K(1,1)     K(1,0)
       /  \         /   \         /   \
     /      \     /       \     /       \
K(0,2)  K(0,1)  K(0,1)  K(0,0)  K(0,1)   K(0,0)
Recursion tree for Knapsack capacity 2 units and 3 items of 1 unit weight.
```

Since suproblems are evaluated again, this problem has Overlapping Subprolems property. So the 0-1 Knapsack problem has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array K[][] in bottom up manner. Following is Dynamic Programming based implementation.

C++

```
// A Dynamic Programming based solution for 0-1 Knapsack problem
#include<stdio.h>

// A utility function that returns maximum of two integers
int max(int a, int b) { return (a > b)? a : b; }

// Returns the maximum value that can be put in a knapsack of capacity W
int knapSack(int W, int wt[], int val[], int n)
```

```
{
   int i, w;
   int K[n+1][W+1];

   // Build table K[][] in bottom up manner
   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];
}

int main()
{
    int val[] = {60, 100, 120};
    int wt[] = {10, 20, 30};
    int  W = 50;
    int n = sizeof(val)/sizeof(val[0]);
    printf("%d", knapSack(W, wt, val, n));
    return 0;
}
```

Java

```
// A Dynamic Programming based solution for 0-1 Knapsack problem
class Knapsack
{

    // A utility function that returns maximum of two integers
    static int max(int a, int b) { return (a > b)? a : b; }

   // Returns the maximum value that can be put in a knapsack of capacity W
    static int knapSack(int W, int wt[], int val[], int n)
    {
         int i, w;
     int K[][] = new int[n+1][W+1];

     // Build table K[][] in bottom up manner
     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];
    }
```

```java
    // Driver program to test above function
    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));
    }
}
/*This code is contributed by Rajat Mishra */
```

Python

```python
# A Dynamic Programming based Python Program for 0-1 Knapsack problem
# Returns the maximum value that can be put in a knapsack of capacity W
def knapSack(W, wt, val, n):
    K = [[0 for x in range(W+1)] for x in range(n+1)]

    # Build table K[][] in bottom up manner
    for i in range(n+1):
        for w in range(W+1):
            if i==0 or w==0:
                K[i][w] = 0
            elif 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 program to test above function
val = [60, 100, 120]
wt = [10, 20, 30]
W = 50
n = len(val)
print(knapSack(W, wt, val, n))

# This code is contributed by Bhavya Jain
```

C#

```csharp
// A Dynamic Programming based solution for
// 0-1 Knapsack problem
using System;

class GFG {

    // A utility function that returns
    // maximum of two integers
    static int max(int a, int b)
    {
        return (a > b) ? a : b;
    }

    // Returns the maximum value that
    // can be put in a knapsack of
    // capacity W
    static int knapSack(int W, int []wt,
                          int []val, int n)
    {
        int i, w;
```

```
        int [,]K = new int[n+1,W+1];

        // Build table K[][] in bottom
        // up manner
        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] = Math.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
    static void Main()
    {
        int []val = new int[]{60, 100, 120};
        int []wt = new int[]{10, 20, 30};
        int W = 50;
        int n = val.Length;

        Console.WriteLine(
                  knapSack(W, wt, val, n));
    }
}

// This code is contributed by Sam007
```

PHP

```
<?php
// A Dynamic Programming based solution
// for 0-1 Knapsack problem

// Returns the maximum value that
// can be put in a knapsack of
// capacity W
function knapSack($W, $wt, $val, $n)
{

    $K = array(array());

    // Build table K[][] in
    // bottom up manner
    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
    $val = array(60, 100, 120);
    $wt = array(10, 20, 30);
    $W = 50;
    $n = count($val);
    echo knapSack($W, $wt, $val, $n);

// This code is contributed by Sam007.
?>
```

Output:

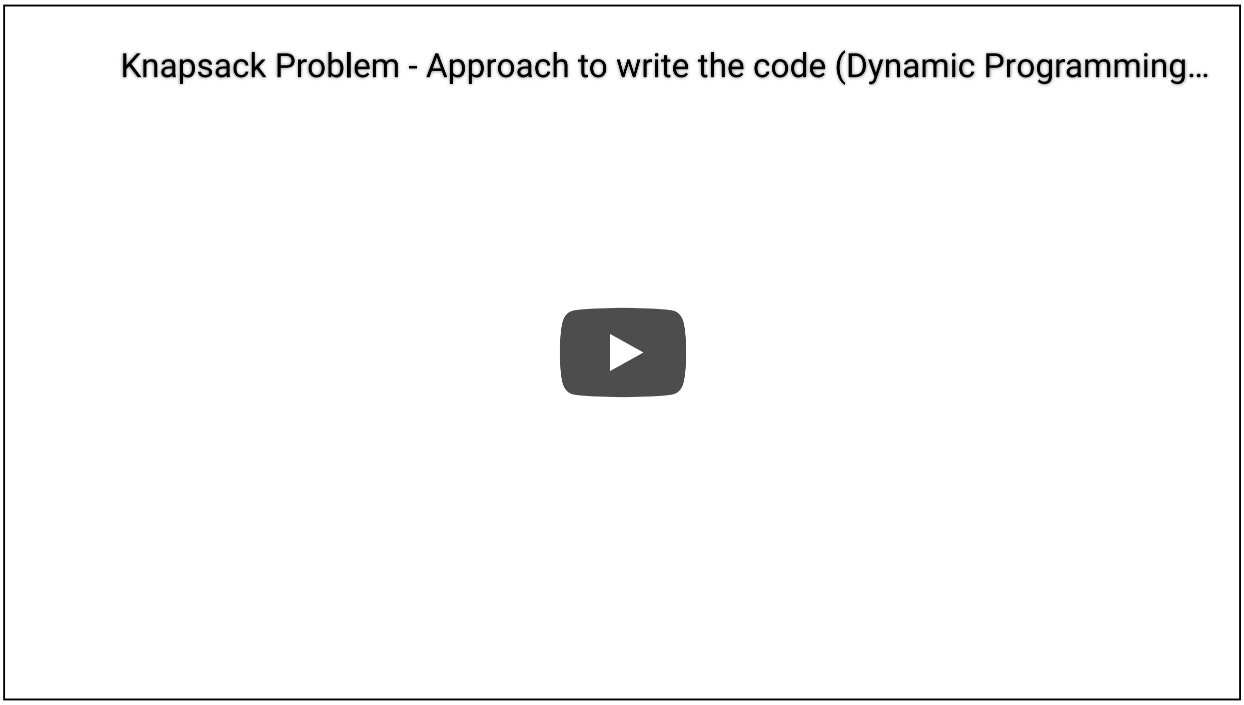
```
220
```

Time Complexity: O(nW) where n is the number of items and W is the capacity of knapsack.

References:

http://www.es.ele.tue.nl/education/5MC10/Solutions/knapsack.pdf

http://www.cse.unl.edu/~goddard/Courses/CSCE310J/Lectures/Lecture8-DynamicProgramming.pdf



Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Recommended Posts:

Fractional Knapsack Problem
Java Program 0-1 Knapsack Problem
Python Program for 0-1 Knapsack Problem
A Space Optimized DP solution for 0-1 Knapsack Problem
0-1 knapsack queries
Knapsack with large Weights
Fractional Knapsack Queries
Printing Items in 0/1 Knapsack
Unbounded Fractional Knapsack
0/1 Knapsack using Branch and Bound
Implementation of 0/1 Knapsack using Branch and Bound
Double Knapsack | Dynamic Programming
Unbounded Knapsack (Repetition of items allowed)
Nuts & Bolts Problem (Lock & Key problem) | Set 2 (Hashmap)
Nuts & Bolts Problem (Lock & Key problem) | Set 1

Improved By : Sam007, rathbhupendra

Article Tags : Dynamic Programming knapsack MakeMyTrip Snapdeal Visa Zoho

Practice Tags : Zoho Snapdeal MakeMyTrip Visa Dynamic Programming