


[Array](#) [Matrix](#) [Strings](#) [Hashing](#) [Linked List](#) [Stack](#) [Queue](#) [Binary Tree](#) [Binary Search Tre](#)

0-1 Knapsack Problem | DP-10

Difficulty Level : Medium • Last Updated : 02 Jun, 2022

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

0-1 Knapsack Problem

value[] = {60, 100, 120};
 weight[] = {10, 20, 30};
 W = 50;

Solution: 220

Weight = 10; Value = 60;
 Weight = 20; Value = 100;
 Weight = 30; Value = 120;
 Weight = (20+10); Value = (100+60);
 Weight = (30+10); Value = (120+60);
 Weight = (30+20); Value = (120+100);
 Weight = (30+20+10) > 50

Recommended Problem

0 - 1 Knapsack Problem

Dynamic Programming [Amazon](#) [Directi](#) +9 more

Solve Problem

Submission count: 1.6L

Start Your Coding Journey Now!

[Login](#)
[Register](#)

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.

Optimal Sub-structure: To consider all subsets of items, there can be two cases for every item.

1. **Case 1:** The item is included in the optimal subset.
2. **Case 2:** The item is not included in the optimal set.

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

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

If the weight of ' n th' item is greater than ' W ', then the n th item cannot be included and **Case 1** is the only possibility.

Below is the implementation of the above approach:

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
```



Start Your Coding Journey Now!

[Login](#)
[Register](#)

```

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;

```



Start Your Coding Journey Now!

[Login](#)
[Register](#)

```

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)

```



Start Your Coding Journey Now!

[Login](#)
[Register](#)

```

// 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
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

```
# 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
```



Start Your Coding Journey Now!

[Login](#)
[Register](#)

```

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

#Driver Code
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#

```

/* 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)
    {

```



Start Your Coding Journey Now!

[Login](#)
[Register](#)

```

// 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
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

```



Start Your Coding Journey Now!

[Login](#)
[Register](#)

```

// 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
?>
```

Javascript

```

<script>

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

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

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



Start Your Coding Journey Now!

[Login](#)
[Register](#)

```

    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));
}

let val = [ 60, 100, 120 ];
let wt = [ 10, 20, 30 ];
let W = 50;
let n = val.length;

document.write(knapSack(W, wt, val, n));

</script>

```

Output

220

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

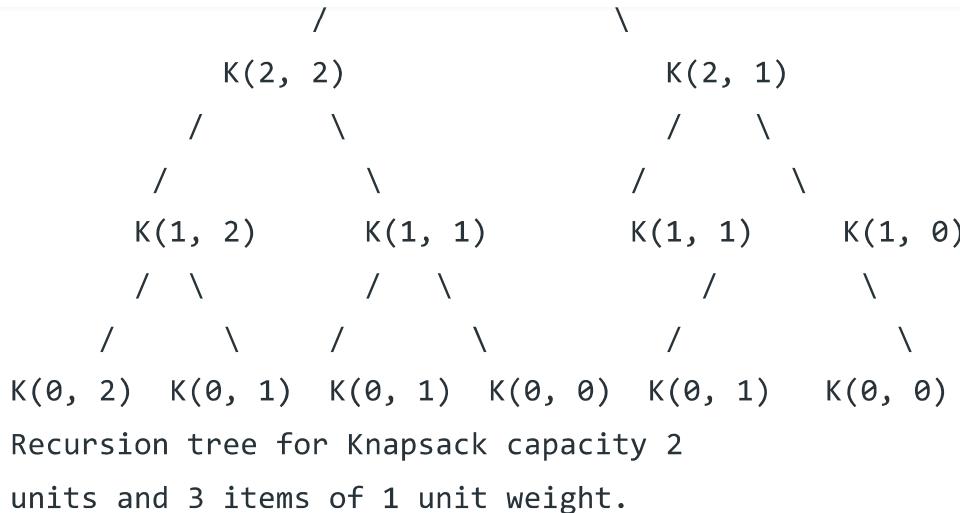
In the following recursion tree, K() refers to knapSack(). The two parameters indicated in the following recursion tree are n and W.

The recursion tree is for following sample inputs.

wt[] = {1, 1, 1}, W = 2, val[] = {10, 20, 30}

K(n, W)

Start Your Coding Journey Now!

[Login](#)
[Register](#)


Complexity Analysis:

- **Time Complexity:** $O(2^n)$.

As there are redundant subproblems.

- **Auxiliary Space :** $O(1) + O(N)$.

As no extra data structure has been used for storing values but $O(N)$ auxiliary stack space(ASS) has been used for recursion stack.

Since subproblems are evaluated again, this problem has Overlapping Sub-problems property. So the 0-1 Knapsack problem has both properties (see [this](#) and [this](#)) of a dynamic programming problem.

Method 2: Like other typical [Dynamic Programming\(DP\)_problems](#), re-computation of same subproblems can be avoided by constructing a temporary array $K[][]$ in bottom-up manner. Following is Dynamic Programming based implementation.

Approach: In the Dynamic programming we will work considering the same cases as mentioned in the recursive approach. In a $DP[][]$ table let's consider all the possible weights from '1' to 'W' as the columns and weights that can be kept as the rows.

The state $DP[i][j]$ will denote maximum value of 'j-weight' considering all values from '1' to i^{th} . So if we consider ' w_i ' (weight in ' i^{th} ' row) we can fill it in all columns which have 'weight values $> w_i$ '. Now two possibilities can take place:

 Fill ' w_i ' in the given column.

Do not fill ' w_i ' in the given column.

Start Your Coding Journey Now!

[Login](#)
[Register](#)

weight, $DP[i][j]$ will be equal to the value of ' w_i ' + value of the column weighing ' $j-w_i$ ' in the previous row. So we take the maximum of these two possibilities to fill the current state. This visualisation will make the concept clear:

Let weight elements = {1, 2, 3}

Let weight values = {10, 15, 40}

Capacity=6

0	1	2	3	4	5	6
0	0	0	0	0	0	0
1	0	10	10	10	10	10
2	0	10	15	25	25	25
3	0					

Explanation:

For filling 'weight = 2' we come across ' $j = 3$ ' in which

we take maximum of

$$(10, 15 + DP[1][3-2]) = 25$$

|

|

'2' '2 filled'

not filled

0	1	2	3	4	5	6
0	0	0	0	0	0	0
1	0	10	10	10	10	10
2	0	10	15	25	25	25



Start Your Coding Journey Now!

[Login](#)
[Register](#)

Explanation:

For filling 'weight=3',
 we come across 'j=4' in which
 we take maximum of (25, 40 + DP[2][4-3])
 = 50

For filling 'weight=3'
 we come across 'j=5' in which
 we take maximum of (25, 40 + DP[2][5-3])
 = 55

For filling 'weight=3'
 we come across 'j=6' in which
 we take maximum of (25, 40 + DP[2][6-3])
 = 65

C++

```
// A dynamic programming based
// solution for 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)
{
    int i, w;
    vector<vector<int>> K(n + 1, vector<int>(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] > w)
                K[i][w] = K[i - 1][w];
            else
                K[i][w] = max(K[i - 1][w], K[i - 1][w - wt[i]] + val[i]);
}
```



Start Your Coding Journey Now!

[Login](#)
[Register](#)

```

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
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 Debojyoti Mandal

```

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)
{

```



Start Your Coding Journey Now!

[Login](#)
[Register](#)

```
// 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
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
```



Start Your Coding Journey Now!

[Login](#)
[Register](#)

```

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];
}

// Driver code
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

```

# 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)]

```

Start Your Coding Journey Now!

[Login](#)
[Register](#)

```

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 code
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#

```

// 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];
    }
}

```



Start Your Coding Journey Now!

[Login](#)
[Register](#)

```

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)
{

```

Start Your Coding Journey Now!

[Login](#)
[Register](#)

```

// 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.
?>
```

Javascript

```

<script>
    // A Dynamic Programming based solution
    // for 0-1 Knapsack problem

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

    // Returns the maximum value that can
```



Start Your Coding Journey Now!

[Login](#)
[Register](#)

```

let i, w;
let K = new Array(n + 1);

// Build table K[][] in bottom up manner
for (i = 0; i <= n; i++)
{
    K[i] = new Array(W + 1);
    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];
}

let val = [ 60, 100, 120 ];
let wt = [ 10, 20, 30 ];
let W = 50;
let n = val.length;
document.write(knapSack(W, wt, val, n));
</script>

```

Output

220

Complexity Analysis:

- **Time Complexity:** $O(N*W)$.

where 'N' is the number of weight element and 'W' is capacity. As for every weight element we traverse through all weight capacities $1 \leq w \leq W$.

- **Auxiliary Space:** $O(N*W)$.

The use of 2-D array of size ' $N*W$ '.

Start Your Coding Journey Now!

[Login](#)
[Register](#)

C++

```
#include <bits/stdc++.h>
using namespace std;

// we can further improve the above Knapsack function's space
// complexity
int knapSack(int W, int wt[], int val[], int n)
{
    int i, w;
    int K[2][W + 1];
    // We know we are always using the current row or
    // the previous row of the array/vector . Thereby we can
    // improve it further by using a 2D array but with only
    // 2 rows i%2 will be giving the index inside the bounds
    // of 2d array K

    for (i = 0; i <= n; i++) {
        for (w = 0; w <= W; w++) {
            if (i == 0 || w == 0)
                K[i % 2][w] = 0;
            else if (wt[i - 1] <= w)
                K[i % 2][w] = max(
                    val[i - 1]
                    + K[(i - 1) % 2][w - wt[i - 1]],
                    K[(i - 1) % 2][w]);
            else
                K[i % 2][w] = K[(i - 1) % 2][w];
        }
    }
    return K[n % 2][W];
}

// 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;
}
```



Start Your Coding Journey Now!

[Login](#)[Register](#)

Java

```
import java.util.*;
class GFG {

    // we can further improve the above Knapsack function's space
    // complexity
    static int knapSack(int W, int wt[], int val[], int n)
    {
        int i, w;
        int [][]K = new int[2][W + 1];

        // We know we are always using the current row or
        // the previous row of the array/vector . Thereby we can
        // improve it further by using a 2D array but with only
        // 2 rows i%2 will be giving the index inside the bounds
        // of 2d array K
        for (i = 0; i <= n; i++) {
            for (w = 0; w <= W; w++) {
                if (i == 0 || w == 0)
                    K[i % 2][w] = 0;
                else if (wt[i - 1] <= w)
                    K[i % 2][w] = Math.max(
                        val[i - 1]
                        + K[(i - 1) % 2][w - wt[i - 1]],
                        K[(i - 1) % 2][w]);
                else
                    K[i % 2][w] = K[(i - 1) % 2][w];
            }
        }
        return K[n % 2][W];
    }

    // Driver Code
    public static void main(String[] args)
    {
        int val[] = { 60, 100, 120 };
        int wt[] = { 10, 20, 30 };
        int W = 50;
        int n = val.length;

        System.out.print(knapSack(W, wt, val, n));
    }
}
```



Start Your Coding Journey Now!

[Login](#)
[Register](#)

Python3

```
# we can further improve the above Knapsack function's space
# complexity
def knapSack(W, wt, val, n):

    K = [[0 for x in range(W+1)] for y in range(2)]

    # We know we are always using the current row or
    # the previous row of the array/vector . Thereby we can
    # improve it further by using a 2D array but with only
    # 2 rows i%2 will be giving the index inside the bounds
    # of 2d array K
    for i in range(n + 1):
        for w in range(W + 1):
            if (i == 0 or w == 0):
                K[i % 2][w] = 0
            elif (wt[i - 1] <= w):
                K[i % 2][w] = max(
                    val[i - 1]
                    + K[(i - 1) % 2][w - wt[i - 1]],
                    K[(i - 1) % 2][w])
            else:
                K[i % 2][w] = K[(i - 1) % 2][w]

    return K[n % 2][W]

# Driver Code
if __name__ == "__main__":

    val = [60, 100, 120]
    wt = [10, 20, 30]
    W = 50
    n = len(val)

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

    # This code is contributed by ukasp.
```

C#

Start Your Coding Journey Now!

[Login](#)
[Register](#)

```
// we can further improve the above Knapsack function's space
// complexity
static int knapSack(int W, int []wt, int []val, int n) {
    int i, w;
    int[,] K = new int[2,W + 1];

    // We know we are always using the current row or
    // the previous row of the array/vector . Thereby we can
    // improve it further by using a 2D array but with only
    // 2 rows i%2 will be giving the index inside the bounds
    // of 2d array K
    for (i = 0; i <= n; i++) {
        for (w = 0; w <= W; w++) {
            if (i == 0 || w == 0)
                K[i % 2, w] = 0;
            else if (wt[i - 1] <= w)
                K[i % 2,w] = Math.Max(val[i - 1] + K[(i - 1) % 2,w - wt[i - 1]]);
            else
                K[i % 2,w] = K[(i - 1) % 2,w];
        }
    }
    return K[n % 2,W];
}

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

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

// This code is contributed by umadevi9616
```

Javascript



```
script>
// we can further improve the above Knapsack function's space
// complexity
```

Start Your Coding Journey Now!

[Login](#)
[Register](#)

```

// We know we are always using the current row or
// the previous row of the array/vector . Thereby we can
// improve it further by using a 2D array but with only
// 2 rows i%2 will be giving the index inside the bounds
// of 2d array K
for (i = 0; i <= n; i++) {
    for (w = 0; w <= W; w++) {
        if (i == 0 || w == 0)
            K[i % 2][w] = 0;
        else if (wt[i - 1] <= w)
            K[i % 2][w] = Math.max(val[i - 1] +
                K[(i - 1) % 2][w - wt[i - 1]],
                K[(i - 1) % 2][w]);
        else
            K[i % 2][w] = K[(i - 1) % 2][w];
    }
}
return K[n % 2][W];
}

// Driver Code
var val = [ 60, 100, 120 ];
var wt = [ 10, 20, 30 ];
var W = 50;
var n = val.length;

document.write(knapSack(W, wt, val, n));

// This code is contributed by Rajput-Ji
</script>

```

Complexity Analysis:

- **Time Complexity:** $O(N*W)$.
- **Auxiliary Space:** $O(2*W)$

As we are using a 2-D array but with only 2 rows.

Method 3: This method uses Memoization Technique (an extension of recursive approach).

This method is basically an extension to the recursive approach so that we can overcome the problem of calculating redundant cases and thus increased complexity. We can solve

Start Your Coding Journey Now!

[Login](#)
[Register](#)

calculating it in exponential complexity we can directly return its result stored in the table in constant time. This method gives an edge over the recursive approach in this aspect.

C++

```
// Here is the top-down approach of
// dynamic programming
#include <bits/stdc++.h>
using namespace std;

// Returns the value of maximum profit
int knapSackRec(int W, int wt[],
                  int val[], int i,
                  int** dp)
{
    // base condition
    if (i < 0)
        return 0;
    if (dp[i][W] != -1)
        return dp[i][W];

    if (wt[i] > W) {

        // Store the value of function call
        // stack in table before return
        dp[i][W] = knapSackRec(W, wt,
                               val, i - 1,
                               dp);
        return dp[i][W];
    }
    else {
        // Store value in a table before return
        dp[i][W] = max(val[i]
                       + knapSackRec(W - wt[i],
                                      wt, val,
                                      i - 1, dp),
                      knapSackRec(W, wt, val,
                                  i - 1, dp));
    }

    // Return value of table after storing
    return dp[i][W];
}
```



Start Your Coding Journey Now!

[Login](#)
[Register](#)

```
// double pointer to declare the
// table dynamically
int** dp;
dp = new int*[n];

// loop to create the table dynamically
for (int i = 0; i < n; i++)
    dp[i] = new int[W + 1];

// loop to initially filled the
// table with -1
for (int i = 0; i < n; i++)
    for (int j = 0; j < W + 1; j++)
        dp[i][j] = -1;
return knapSackRec(W, wt, val, n - 1, dp);
}

// 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;
}
```

Java

```
// Here is the top-down approach of
// dynamic programming
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 value of maximum profit
static int knapSackRec(int W, int wt[],
                      int val[], int n,
                      int [][]dp)
```



Start Your Coding Journey Now!

[Login](#)
[Register](#)

```

if (n == 0 || w == 0)
    return 0;

if (dp[n][w] != -1)
    return dp[n][w];

if (wt[n - 1] > w)

    // Store the value of function call
    // stack in table before return
    return dp[n][w] = knapSackRec(w, wt, val,
                                    n - 1, dp);

else

    // Return value of table after storing
    return dp[n][w] = max((val[n - 1] +
                           knapSackRec(w - wt[n - 1], wt,
                                         val, n - 1, dp)),
                           knapSackRec(w, wt, val,
                                         n - 1, dp));
}

static int knapSack(int w, int wt[], int val[], int N)
{

    // Declare the table dynamically
    int dp[][] = new int[N + 1][w + 1];

    // Loop to initially filled the
    // table with -1
    for(int i = 0; i < N + 1; i++)
        for(int j = 0; j < w + 1; j++)
            dp[i][j] = -1;

    return knapSackRec(w, wt, val, N, dp);
}

// Driver Code
public static void main(String [] args)
{
    int val[] = { 60, 100, 120 };
    int wt[] = { 10, 20, 30 };

    int W = 50;
    int N = val.length;
}

```



Start Your Coding Journey Now!

[Login](#)[Register](#)

// This Code is contributed By FARAZ AHMAD

Python3

```
# This is the memoization approach of
# 0 / 1 Knapsack in Python in simple
# we can say recursion + memoization = DP

# driver code
val = [60, 100, 120 ]
wt = [10, 20, 30 ]
W = 50
n = len(val)

# We initialize the matrix with -1 at first.
t = [[-1 for i in range(W + 1)] for j in range(n + 1)]

def knapsack(wt, val, W, n):

    # base conditions
    if n == 0 or W == 0:
        return 0
    if t[n][W] != -1:
        return t[n][W]

    # choice diagram code
    if wt[n-1] <= W:
        t[n][W] = max(
            val[n-1] + knapsack(
                wt, val, W-wt[n-1], n-1),
            knapsack(wt, val, W, n-1))
        return t[n][W]
    elif wt[n-1] > W:
        t[n][W] = knapsack(wt, val, W, n-1)
        return t[n][W]

print(knapsack(wt, val, W, n))

# This code is contributed by Prosun Kumar Sarkar
```



Start Your Coding Journey Now!

[Login](#)[Register](#)

```
// Here is the top-down approach of
// dynamic programming
using System;
public 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 value of maximum profit
    static int knapSackRec(int W, int[] wt, int[] val,
                          int n, int[, ] dp)
    {
        // Base condition
        if (n == 0 || W == 0)
            return 0;
        if (dp[n, W] != -1)
            return dp[n, W];
        if (wt[n - 1] > W)

            // Store the value of function call
            // stack in table before return
            return dp[n, W]
                = knapSackRec(W, wt, val, n - 1, dp);

        else

            // Return value of table after storing
            return dp[n, W]
                = max((val[n - 1]
                    + knapSackRec(W - wt[n - 1], wt, val,
                                  n - 1, dp)),
                    knapSackRec(W, wt, val, n - 1, dp));
    }

    static int knapSack(int W, int[] wt, int[] val, int N)
    {
        // Declare the table dynamically
        int[, ] dp = new int[N + 1, W + 1];

        // Loop to initially filled the
        // table with -1
        for (int i = 0; i < N + 1; i++)
    }
```

Start Your Coding Journey Now!

[Login](#)
[Register](#)

```

        return knapSackRec(W, wt, val, N, dp);
    }

    // Driver Code
    static public 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 Dharanendra L V.

```

Javascript

```

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

// Returns the value of maximum profit
function knapSackRec(W, wt, val, n, dp)
{

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

    if (dp[n][W] != -1)
        return dp[n][W];

    if (wt[n - 1] > W)

        // Store the value of function call
        // stack in table before return
        return dp[n][W] = knapSackRec(W, wt, val,

```



Start Your Coding Journey Now!

[Login](#)
[Register](#)

```
// Return value of table after storing
return dp[n][W] = max((val[n - 1] +
    knapSackRec(W - wt[n - 1], wt,
                  val, n - 1, dp)),
    knapSackRec(W, wt, val,
                  n - 1, dp));
}

function knapSack( W, wt, val, N)
{
    // Declare the table dynamically
    var dp = new Array(N + 1);
    for(var i = 0; i < dp.length; i++)
    {
        dp[i]=new Array(W + 1);
    }

    // Loop to initially filled the
    // table with -1
    for(var i = 0; i < N + 1; i++)
        for(var j = 0; j < W + 1; j++)
            dp[i][j] = -1;

    return knapSackRec(W, wt, val, N, dp);
}

var val= [ 60, 100, 120 ];
var wt = [ 10, 20, 30 ];

var W = 50;
var N = val.length;

document.write(knapSack(W, wt, val, N));

// This code is contributed by akshitsaxenaa09.

</script>
```

Output

20

Start Your Coding Journey Now!

[Login](#)
[Register](#)

As redundant calculations of states are avoided.

- **Auxiliary Space:** $O(N*W) + O(N)$.

The use of 2D array data structure for storing intermediate states and $O(N)$ auxiliary stack space (ASS) has been used for recursion stack:

[Note: For 32bit integer use long instead of int.]

References:

- <http://www.es.ele.tue.nl/education/5MC10/Solutions/knapsack.pdf>
- <http://www.cse.unl.edu/~goddard/Courses/CSCE310J/Lectures/Lecture8-DynamicProgramming.pdf>
- https://youtu.be/T4bY72lCQac?list=PLqM7alHXFySGMu2CSdW_6d2u1o6WFTI0-

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

Method 4 :- Again we use the dynamic programming approach with even more optimized space complexity .

C++

```
#include <bits/stdc++.h>
using namespace std;
int knapSack(int W, int wt[], int val[], int n)
{
    // making and initializing dp array
    int dp[W + 1];
    memset(dp, 0, sizeof(dp));

    for (int i = 1; i < n + 1; i++) {
        for (int w = W; w >= 0; w--) {

            if (wt[i - 1] <= w)
                // finding the maximum value
                dp[w] = max(dp[w],
                            dp[w - wt[i - 1]] + val[i - 1]);
        }
    }
}
```



Start Your Coding Journey Now!

[Login](#)[Register](#)

```
{  
    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;  
}
```

Java

```
import java.util.*;  
  
class GFG{  
    static int knapSack(int W, int wt[], int val[], int n)  
    {  
        // making and initializing dp array  
        int []dp = new int[W + 1];  
  
        for (int i = 1; i < n + 1; i++) {  
            for (int w = W; w >= 0; w--) {  
  
                if (wt[i - 1] <= w)  
  
                    // finding the maximum value  
                    dp[w] = Math.max(dp[w],  
                                     dp[w - wt[i - 1]] + val[i - 1]);  
            }  
        }  
        return dp[W]; // returning the maximum value of knapsack  
    }  
  
    // Driver code  
    public static void main(String[] args)  
    {  
        int val[] = { 60, 100, 120 };  
        int wt[] = { 10, 20, 30 };  
        int W = 50;  
        int n = val.length;  
        System.out.print(knapSack(W, wt, val, n));  
    }  
  
    // This code is contributed by gauravrajput1
```

Start Your Coding Journey Now!

[Login](#)
[Register](#)

```
# code
# 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):
    dp = [0 for i in range(W+1)] # Making the dp array

    for i in range(1, n+1): # taking first i elements
        for w in range(W, 0, -1): # starting from back, so that we also have data of
            # previous computation when taking i-1 items
            if wt[i-1] <= w:
                # finding the maximum value
                dp[w] = max(dp[w], dp[w-wt[i-1]]+val[i-1])

    return dp[W] # returning the maximum value of knapsack

# Driver code
val = [60, 100, 120]
wt = [10, 20, 30]
W = 50
n = len(val)
# This code is contributed by Suyash Saxena
print(knapSack(W, wt, val, n))
```

C#

```
using System;
public class GFG {
    static int knapSack(int W, int []wt, int []val, int n)
    {

        // making and initializing dp array
        int[] dp = new int[W + 1];

        for (int i = 1; i < n + 1; i++)
        {
            for (int w = W; w >= 0; w--)
            {
```



Start Your Coding Journey Now!

[Login](#)
[Register](#)

```

        // finding the maximum value
        dp[w] = Math.Max(dp[w], dp[w - wt[i - 1]] + val[i - 1]);
    }
}
return dp[W]; // returning the maximum value of knapsack
}

// Driver code
public static void Main(String[] args) {
    int []val = { 60, 100, 120 };
    int []wt = { 10, 20, 30 };
    int W = 50;
    int n = val.Length;
    Console.WriteLine(knapSack(W, wt, val, n));
}
}

// This code is contributed by gauravrajput1

```

Javascript

```

<script>
function knapSack(W , wt , val , n)
{
    // making and initializing dp array
    var dp = Array(W + 1).fill(0);

    for (i = 1; i < n + 1; i++) {
        for (w = W; w >= 0; w--) {

            if (wt[i - 1] <= w)

                // finding the maximum value
                dp[w] = Math.max(dp[w], dp[w - wt[i - 1]] + val[i - 1]);
        }
    }
    return dp[W]; // returning the maximum value of knapsack
}

// Driver code
var val = [ 60, 100, 120 ];
var wt = [ 10, 20, 30 ];
var W = 50;
var n = val.length;

```



Start Your Coding Journey Now!

[Login](#)[Register](#)

</script>

Output

220

Complexity Analysis:

Time Complexity: $O(N*W)$. As redundant calculations of states are avoided.

Auxiliary Space: $O(W)$ As we are using 1-D array instead of 2-D array.

Like 537

[Previous](#)[Next](#)

Start Your Coding Journey Now!

[Login](#)
[Register](#)

- 01** Java Program 0-1 Knapsack Problem

19, Mar 12

- 02** Python Program for 0-1 Knapsack Problem

19, Mar 12

- 03** 0/1 Knapsack Problem to print all possible solutions

07, May 20

- 04** C++ Program for the Fractional Knapsack Problem

18, Jul 20

- 05** A Space Optimized DP solution for 0-1 Knapsack Problem

31, Aug 16

- 06** Extended Knapsack Problem

31, May 20

- 07** Fractional Knapsack Problem

23, Mar 16

- 08** Implementation of 0/1 Knapsack using Branch and Bound

29, Apr 16

Article Contributed By :



GeeksforGeeks

Vote for difficulty

Current difficulty : [Medium](#)

Improved By :

Sam007, rathbhupendra, MohammadMudassir, bidibaaz123, prosunsarkar7, ssatyanand7, aaku34, uchiha1101, fa6879, believer411, dharanendralv23, sahityakmr, namdeoanuj15052001, jyoti369, decode2207, suresh07, suyashsincever13, akshitsaxenaa09, moumenhamada30, elbradey8,

Start Your Coding Journey Now!

[Login](#)
[Register](#)

Article Tags : knapsack, MakeMyTrip, Snapdeal, Visa, Zoho, Dynamic Programming

Practice Tags : MakeMyTrip, Snapdeal, Visa, Zoho, Dynamic Programming

[Improve Article](#)
[Report Issue](#)

Writing code in comment? Please use [ide.geeksforgeeks.org](#), generate link and share the link here.

[Load Comments](#)


A-143, 9th Floor, Sovereign Corporate Tower,
Sector-136, Noida, Uttar Pradesh - 201305

feedback@geeksforgeeks.org

Company

[About Us](#)

[Careers](#)

[In Media](#)

[Contact Us](#)

[Privacy Policy](#)

Learn

[Algorithms](#)

[Data Structures](#)

[SDE Cheat Sheet](#)

[Machine learning](#)

[CS Subjects](#)



Start Your Coding Journey Now!

[Login](#)[Register](#)

News

- Top News
- Technology
- Work & Career
- Business
- Finance
- Lifestyle
- Knowledge

Languages

- Python
- Java
- CPP
- Golang
- C#
- SQL
- Kotlin

Web Development

- Web Tutorials
- Django Tutorial
- HTML
- JavaScript
- Bootstrap
- ReactJS
- NodeJS

Contribute

- Write an Article
- Improve an Article
- Pick Topics to Write
- Write Interview Experience
- Internships
- Video Internship

@geeksforgeeks , Some rights reserved

