



## 0/1 Knapsack Problem

**Prerequisites:** [Introduction to Knapsack Problem, its Types and How to solve them](#)

Given **N** items where each item has some weight and profit associated with it and also given a bag with capacity **W**, [i.e., the bag can hold at most **W** weight in it]. The task is to put the items into the bag such that the sum of profits associated with them is the maximum possible.

**Note:** The constraint here is we can either put an item completely into the bag or cannot put it at all [It is not possible to put a part of an item into the bag].

**Examples:**

**Example 1:**  $N = 3, W = 4, \text{profit}[] = \{1, 2, 3\}, \text{weight}[] = \{4, 5, 1\}$

**Input:**  $N = 3, W = 4, \text{profit}[] = \{1, 2, 3\}, \text{weight}[] = \{4, 5, 1\}$

**Output:** 3

**Explanation:** There are two items which have weight less than or equal to 4. If we select the item with weight 4, the possible profit is 1. And if



*maximum possible profit is 3. Note that we cannot put both the items with weight 4 and 1 together as the capacity of the bag is 4.*

**Input:**  $N = 3$ ,  $W = 3$ ,  $profit[] = \{1, 2, 3\}$ ,  $weight[] = \{4, 5, 6\}$

**Output:** 0

Recommended Problem

## 0 - 1 Knapsack Problem

Solve Problem

Dynamic Programming Algorithms [Flipkart](#) [Morgan Stanley](#) [+9 more](#)

Submission count: 3.6L

## Recursion Approach for 0/1 Knapsack Problem:

To solve the problem follow the below idea:



*A simple solution is to consider all subsets of items and calculate the total weight and profit of all subsets. Consider the only subsets whose total weight is smaller than  $W$ . From all such subsets, pick the subset with maximum profit.*

**Optimal Substructure:** *To consider all subsets of items, there can be two cases for every item.*

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

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#).

Follow the below steps to solve the problem:

The maximum value obtained from 'N' items is the max of the following two values.

- Case 1 (include the  $N^{\text{th}}$  item): Value of the  $N^{\text{th}}$  item plus maximum value obtained by remaining  $N-1$  items and remaining weight i.e. ( $W$ -weight of the  $N^{\text{th}}$  item).
- Case 2 (exclude the  $N^{\text{th}}$  item): Maximum value obtained by  $N-1$  items and  $W$  weight.
- If the weight of the ' $N^{\text{th}}$ ' item is greater than ' $W$ ', then the  $N^{\text{th}}$  item cannot be included and **Case 2** 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
    // 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);
```

```

    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 profit[] = { 60, 100, 120 };
    int weight[] = { 10, 20, 30 };
    int W = 50;
    int n = sizeof(profit) / sizeof(profit[0]);
    cout << knapSack(W, weight, profit, 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 max(
        val[n - 1]
        + knapSack(W - wt[n - 1], wt, val, n - 1),
        knapSack(W, wt, val, n - 1));
}

// Driver code
int main()
{
    int profit[] = { 60, 100, 120 };
    int weight[] = { 10, 20, 30 };
    int W = 50;
    int n = sizeof(profit) / sizeof(profit[0]);
    printf("%d", knapSack(W, weight, profit, 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

```

```

        + knapSack(W - wt[n - 1], wt,
                    val, n - 1),
        knapSack(W, wt, val, n - 1));
    }

    // Driver code
    public static void main(String args[])
    {
        int profit[] = new int[] { 60, 100, 120 };
        int weight[] = new int[] { 10, 20, 30 };
        int W = 50;
        int n = profit.length;
        System.out.println(knapSack(W, weight, profit, 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
    # 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(

```

```
# end of function knapSack

# Driver Code
if __name__ == '__main__':
    profit = [60, 100, 120]
    weight = [10, 20, 30]
    W = 50
    n = len(profit)
    print knapSack(W, weight, profit, 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)
    {

        // 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
```

```

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

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

        Console.WriteLine(knapSack(W, weight, profit, n));
    }
}

// This code is contributed by Sam007

```

## Javascript

```

/* 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)
{
    // 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)

```



```

        // (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 profit = [ 60, 100, 120 ];
    let weight = [ 10, 20, 30 ];
    let W = 50;
    let n = profit.length;

    document.write(knapSack(W, weight, profit, n));

```

## 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),
            $val[$n - 1] +
                knapSack($W, $wt, $val, $n - 1));
}

```

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

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

Learn [Data Structures & Algorithms](#) with GeeksforGeeks

## Output

220

**Time Complexity:**  $O(2^N)$

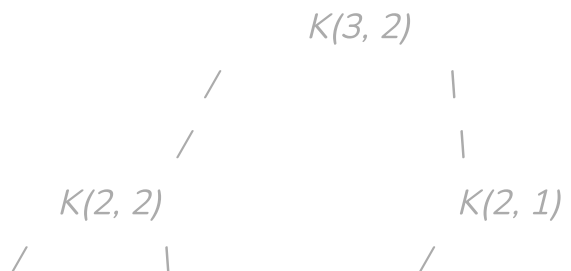
**Auxiliary Space:**  $O(N)$ , Stack space required for recursion

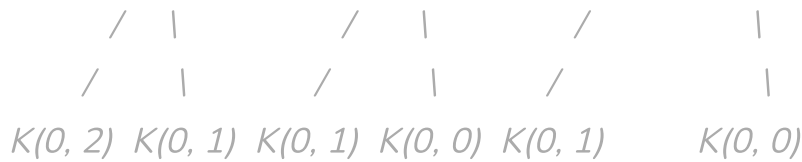
## Dynamic Programming Approach for 0/1 Knapsack Problem

### Memoization Approach for 0/1 Knapsack Problem:

**Note:** It should be noted that the above function using recursion computes the same subproblems again and again. See the following recursion tree,  $K(1, 1)$  is being evaluated twice.

*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.  
 $weight[] = \{1, 1, 1\}$ ,  $W = 2$ ,  $profit[] = \{10, 20, 30\}$*





*Recursion tree for Knapsack capacity 2 units and 3 items of 1 unit weight.*

As there are repetitions of the same subproblem again and again we can implement the following idea to solve the problem.

*If we get a subproblem the first time, we can solve this problem by creating a 2-D array that can store a particular state  $(n, w)$ . Now if we come across the same state  $(n, w)$  again instead of calculating it in exponential complexity we can directly return its result stored in the table in constant time.*

Below is the implementation of the above approach:

## 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 index, int** dp)
{
    // base condition
    if (index < 0)
        return 0;
    if (dp[index][W] != -1)
        return dp[index][W];

    if (wt[index] > W) {

        // Store the value of function call
        // store in table before return
    }
```

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#).

```

    }
    else {
        // Store value in a table before return
        dp[index][W] = max(val[index]
                        + knapSackRec(W - wt[index], wt, val,
                                    index - 1, dp),
                        knapSackRec(W, wt, val, index - 1, dp));

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

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

```

## Java



```

import java.io.*;

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++)
            for (int j = 0; j < W + 1; j++)

```

```

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

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

        int W = 50;
        int N = profit.length;

        System.out.println(knapSack(W, weight, profit, N));
    }
}

// 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

```

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]

# Driver code

```

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#).

```

W = 50
n = len(profit)

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

```

# This code is contributed by Prosun Kumar Sarkar

## C#

```

// 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));
    }
}

```

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#).

```

{

    // 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
static public void Main()
{

    int[] profit = new int[] { 60, 100, 120 };
    int[] weight = new int[] { 10, 20, 30 };

    int W = 50;
    int N = profit.Length;

    Console.WriteLine(knapSack(W, weight, profit, N));
}

}

// This Code is contributed By Dharanendra L V.

```

## Javascript

```

// 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 dp[n][W];

    if (wt[n - 1] > W)

        // Store the value of function call
        // store 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));
}

function knapSack( W, wt,val,N)
{

    // Declare the dp table dynamically
    // Initializing dp tables(row and cols) with -1 below
    var dp = new Array(N+1).fill(-1).map(el => new Array(W+1).fill(-1));
    return knapSackRec(W, wt, val, N, dp);
}

var profit= [ 60, 100, 120 ];
var weight = [ 10, 20, 30 ];

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

document.write(knapSack(W, weight, profit, N));

// This code is contributed by akshitsaxenaa09.

```

Learn [Data Structures & Algorithms](#) with GeeksforGeeks

## Output

220

storing intermediate states and  $O(N)$  auxiliary stack space(ASS) has been used for recursion stack

### Bottom-up Approach for 0/1 Knapsack Problem:

To solve the problem follow the below idea:

*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. Like other typical [Dynamic Programming\(DP\) problems](#), re-computation of the same subproblems can be avoided by constructing a temporary array  $K[]$  in a bottom-up manner.*

### Illustration:

Below is the illustration of the above approach:

*Let,  $weight[] = \{1, 2, 3\}$ ,  $profit[] = \{10, 15, 40\}$ ,  $Capacity = 6$*

- If no element is filled, then the possible profit is 0.*

<i>weight→ item↓/</i>	<i>0</i>	<i>1</i>	<i>2</i>	<i>3</i>	<i>4</i>	<i>5</i>	<i>6</i>
<i>0</i>	<i>0</i>	<i>0</i>	<i>0</i>	<i>0</i>	<i>0</i>	<i>0</i>	<i>0</i>
<i>1</i>							
<i>2</i>							
<i>3</i>							

- **For filling the first item in the bag:** If we follow the above mentioned procedure, the table will look like the following.

weight→ item↓/	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2							
3							

- **For filling the second item:**

When  $jthWeight = 2$ , then maximum possible profit is  $\max(10, DP[1][2-2] + 15) = \max(10, 15) = 15$ .

When  $jthWeight = 3$ , then maximum possible profit is  $\max(2 \text{ not put, } 2 \text{ is put into bag}) = \max(DP[1][3], 15 + DP[1][3-2]) = \max(10, 25) = 25$ .

weight→ item↓/	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0	10	15	25	25	25	25
3							

- **For filling the third item:**

When  $jthWeight = 3$ , the maximum possible profit is  $\max(DP[2][3],$

When  $jthWeight = 4$ , the maximum possible profit is  $\max(DP[2][4], 40 + DP[2][4-3]) = \max(25, 50) = 50$ .

When  $jthWeight = 5$ , the maximum possible profit is  $\max(DP[2][5], 40 + DP[2][5-3]) = \max(25, 55) = 55$ .

When  $jthWeight = 6$ , the maximum possible profit is  $\max(DP[2][6], 40 + DP[2][6-3]) = \max(25, 65) = 65$ .

<i>weight</i> → <i>item</i> ↓/	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0	10	15	25	25	25	25
3	0	10	15	40	50	55	65

Below is the implementation of the above approach:

## 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, j, ...
```

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#).

```
// 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 profit[] = { 60, 100, 120 };
    int weight[] = { 10, 20, 30 };
    int W = 50;
    int n = sizeof(profit) / sizeof(profit[0]);

    cout << knapSack(W, weight, profit, 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)
{
```

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

## Java

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

import java.io.*;

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)
    {
```

```
// 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 profit[] = new int[] { 60, 100, 120 };
    int weight[] = new int[] { 10, 20, 30 };
    int W = 50;
    int n = profit.length;
    System.out.println(knapSack(W, weight, profit, n));
}

/*This code is contributed by Rajat Mishra */
```

## Python3

```
# 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-1][w-wt[i-1]],
        K[i-1][w])

    else:
        K[i][w] = K[i-1][w]

    return K[n][W]

# Driver code
if __name__ == '__main__':
    profit = [60, 100, 120]
    weight = [10, 20, 30]
    W = 50
    n = len(profit)
    print(knapSack(W, weight, profit, 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];

        // 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;
            }
        }
    }
}

```



```

        + 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[] profit = new int[] { 60, 100, 120 };
    int[] weight = new int[] { 10, 20, 30 };
    int W = 50;
    int n = profit.Length;

    Console.WriteLine(knapSack(W, weight, profit, n));
}

// This code is contributed by Sam007

```

## Javascript

```

// 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
// be put in a knapsack of capacity W
function knapSack(W, wt, val, n)
{
    let i, w;
    let K = new Array(n + 1);

    // Build table K[][] in bottom up manner
    for (i = 0; i <= n; i++)

```

```

    {
        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 profit = [ 60, 100, 120 ];
let weight = [ 10, 20, 30 ];
let W = 50;
let n = profit.length;
document.write(knapSack(W, weight, profit, n));

```

## 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 - 1][$w]);
    else
        $K[$i][$w] = $K[$i - 1][$w];
    }
}

return $K[$n][$W];
}

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

// This code is contributed by Sam007.
?>

```

Learn [Data Structures & Algorithms](#) with GeeksforGeeks

## Output

220

**Time Complexity:**  $O(N * W)$ . where 'N' is the number of elements and 'W' is capacity.

**Auxiliary Space:**  $O(N * W)$ . The use of a 2-D array of size 'N\*W'.

## Space optimized Approach for 0/1 Knapsack Problem using Dynamic Programming:

To solve the problem follow the below idea:

*For calculating the current row of the dp[] array we require only previous row, but if we start traversing the rows from right to left then it can be done with a single row only*

Below is the implementation of the above approach:

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#).

## C++

```
// C++ program for the above approach

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

// Function to find the maximum profit
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]);
        }
    }
    // Returning the maximum value of knapsack
    return dp[W];
}

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

## Java

```
// Java program for the above approach
```

```
.....
```

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#).

```

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]);

        }
    }
    // Returning the maximum value of knapsack
    return dp[W];
}

// Driver code
public static void main(String[] args)
{
    int profit[] = { 60, 100, 120 };
    int weight[] = { 10, 20, 30 };
    int W = 50;
    int n = profit.length;
    System.out.print(knapSack(W, weight, profit, n));
}

// This code is contributed by gauravrajput1

```

## Python3

# Python code to implement the above approach

```

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

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

    # Taking first i elements
    for i in range(1, n+1):

```

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#).

```

# previous computation when taking i-1 items
for w in range(W, 0, -1):
    if wt[i-1] <= w:

        # Finding the maximum value
        dp[w] = max(dp[w], dp[w-wt[i-1]]+val[i-1])

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

# Driver code
if __name__ == '__main__':
    profit = [60, 100, 120]
    weight = [10, 20, 30]
    W = 50
    n = len(profit)
    print(knapSack(W, weight, profit, n))

```

# This code is contributed by Suyash Saxena

## C#

```

// Java program for the above approach

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--) {

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

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

```

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

// This code is contributed by gauravrajput1

## Javascript

```
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]);
        }
    }

    // Returning the maximum value of knapsack
    return dp[W];
}

// Driver code
var profit = [ 60, 100, 120 ];
var weight = [ 10, 20, 30 ];
var W = 50;
var n = profit.length;
document.write(knapSack(W, weight, profit, n));

// This code is contributed by Rajput-Ji
```

Learn [Data Structures & Algorithms](#) with GeeksforGeeks

220

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

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

Feeling lost in the world of random DSA topics, wasting time without progress? It's time for a change! Join our DSA course, where we'll guide you on an exciting journey to master DSA efficiently and on schedule.











Ready to dive in? Explore our Free Demo Content and join our DSA course, trusted by over 100,000 geeks!

- [DSA in C++](#)
- [DSA in Java](#)
- [DSA in Python](#)
- [DSA in JavaScript](#)

Last Updated : 11 Oct, 2023

629

## Similar Reads

 Difference between 0/1 Knapsack problem and Fractional Knapsack problem	 0/1 Knapsack Problem to print all possible solutions
 Extended Knapsack Problem	 C++ Program for the Fractional Knapsack Problem
 Introduction to Knapsack Problem, its Types and How to solve them	 GFact   Why doesn't Greedy Algorithm work for 0-1 Knapsack problem?
 <b>Fractional Knapsack Problem</b>	 A Space Optimized DP solution for 0-1 Knapsack Problem
 Java Program 0-1 Knapsack Problem	 Python Program for 0-1 Knapsack Problem

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#).





Mathematical and Geometric Algorithms - Data Structure and Algorithm Tutorials



Learn Data Structures with Javascript | DSA Tutorial



Introduction to Max-Heap - Data Structure and Algorithm Tutorials



Introduction to Set - Data Structure and Algorithm Tutorials



Introduction to Map - Data Structure and Algorithm Tutorials

[Previous](#)

[Next](#)

[Difference between 0/1 Knapsack problem and Fractional Knapsack problem](#)

[Printing Items in 0/1 Knapsack](#)

Article Contributed By :



GeeksforGeeks

Vote for difficulty

Current difficulty : [Medium](#)

Easy

Normal

Medium

Hard

Expert

Improved By : [Sam007](#), [Rajput-Ji](#), [ukasp](#), [GauravRajput1](#), [bidibaaz123](#), [MohammadMudassir](#), [srinam](#), [aaku34](#), [rathbhupendra](#), [ssatyanand7](#), [akshitsaxenaa09](#), [jyoti369](#), [prosunsarkar7](#), [fa6879](#), [janardhansharma2012](#), [dharanendralv23](#), [believer411](#), [uchiha1101](#), [namdeoanuj15052001](#), [sahityakmr](#), [suresh07](#), [umadevi9616](#), [sweetyty](#), [decode2207](#), [sumitgumber28](#), [suyashsincever13](#), [srimanswamy](#), [moumenhamada30](#), [elbradey8](#), [uditsingla2000](#), [animeshdey](#), [aakash verma 2](#), [harendrakumar123](#), [janardansthox](#), [vinay02856](#), [vaibhav\\_gfg](#)

Article Tags : [knapsack](#) , [MakeMyTrip](#) , [Snapdeal](#) , [Visa](#) , [Zoho](#) , [DSA](#) ,

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#).

Practice Tags : [MakeMyTrip](#), [Snapdeal](#), [Visa](#), [Zoho](#), [Dynamic Programming](#)

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

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

[feedback@geeksforgeeks.org](mailto:feedback@geeksforgeeks.org)



## Company

[About Us](#)[Legal](#)[Terms & Conditions](#)[Careers](#)[In Media](#)[Contact Us](#)[Advertise with us](#)[GFG Corporate Solution](#)

## Explore

[Job-A-Thon Hiring Challenge](#)[Hack-A-Thon](#)[GfG Weekly Contest](#)[Offline Classes \(Delhi/NCR\)](#)[DSA in JAVA/C++](#)[Master System Design](#)[Master CP](#)[GeeksforGeeks Videos](#)

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#).

[Apply for Mentor](#)

## Languages

[Python](#)[Java](#)[C++](#)[PHP](#)[GoLang](#)[SQL](#)[R Language](#)[Android Tutorial](#)

## DSA Roadmaps

[DSA for Beginners](#)[Basic DSA Coding Problems](#)[DSA Roadmap by Sandeep Jain](#)[DSA with JavaScript](#)[Top 100 DSA Interview Problems](#)[All Cheat Sheets](#)

## Computer Science

[GATE CS Notes](#)[Operating Systems](#)[Computer Network](#)[Database Management System](#)[Software Engineering](#)[Digital Logic Design](#)[Engineering Maths](#)

## Data Science & ML

## DSA Concepts

[Data Structures](#)[Arrays](#)[Strings](#)[Linked List](#)[Algorithms](#)[Searching](#)[Sorting](#)[Mathematical](#)[Dynamic Programming](#)

## Web Development

[HTML](#)[CSS](#)[JavaScript](#)[Bootstrap](#)[ReactJS](#)[AngularJS](#)[NodeJS](#)[Express.js](#)[Lodash](#)

## Python

[Python Programming Examples](#)[Django Tutorial](#)[Python Projects](#)[Python Tkinter](#)[OpenCV Python Tutorial](#)[Python Interview Question](#)

## DevOps

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#).

Data Science For Beginner	AWS
Machine Learning Tutorial	Docker
Maths For Machine Learning	Kubernetes
Pandas Tutorial	Azure
NumPy Tutorial	GCP
NLP Tutorial	
Deep Learning Tutorial	

## Competitive Programming

Top DSA for CP  
Top 50 Tree Problems  
Top 50 Graph Problems  
Top 50 Array Problems  
Top 50 String Problems  
Top 50 DP Problems  
Top 15 Websites for CP

## System Design

What is System Design  
Monolithic and Distributed SD  
Scalability in SD  
Databases in SD  
High Level Design or HLD  
Low Level Design or LLD  
Crack System Design Round  
System Design Interview Questions

## Interview Corner

Company Wise Preparation  
Preparation for SDE  
Experienced Interviews  
Internship Interviews  
Competitive Programming  
Aptitude Preparation

## Commerce

Accountancy  
Business Studies  
Economics  
Human Resource Management (HRM)  
Management  
Income Tax  
Finance  
Statistics for Economics

## SSC/ BANKING

SSC CGL Syllabus  
SBI PO Syllabus  
SBI Clerk Syllabus  
IBPS PO Syllabus  
IBPS Clerk Syllabus  
Aptitude Questions  
SSC CGL Practice Papers

## GfG School

CBSE Notes for Class 8  
CBSE Notes for Class 9  
CBSE Notes for Class 10  
CBSE Notes for Class 11  
CBSE Notes for Class 12  
English Grammar

## UPSC

Polity Notes  
Geography Notes  
History Notes  
Science and Technology Notes  
Economics Notes  
Important Topics in Ethics  
UPSC Previous Year Papers

## Write & Earn

Write an Article  
Improve an Article  
Pick Topics to Write  
Share your Experiences  
Internships

@GeeksforGeeks, Sanchhaya Education Private Limited, All rights reserved