0/1 Knapsack Problem

Last Updated : 05 Aug, 2024

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:

> ***Input:*** *N = 3, W = 4, profit[] = {1, 2, 3}, 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 we select the item with weight 1, the possible profit is 3. So the 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*

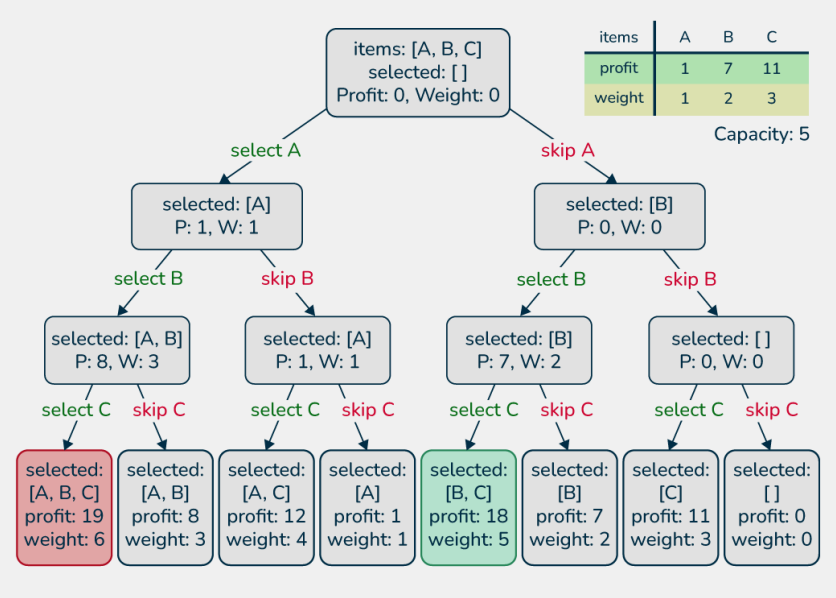
Recursion Approach for 0/1 Knapsack Problem:

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.*



Recursion Tree for 0-1 Knapsack

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^{th} item): Value of the N^{th} item plus maximum value obtained by remaining N-1 items and remaining weight i.e. (W-weight of the N^{th} item).
- Case 2 (exclude the N^{th} item): Maximum value obtained by N-1 items and W weight.

- If the weight of the 'Nth' item is greater than 'W', then the Nth item cannot be included and **Case 2** is the only possibility.

Below is the implementation of the above approach:

C++ C Java Python C# JavaScript PHP

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

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

Output

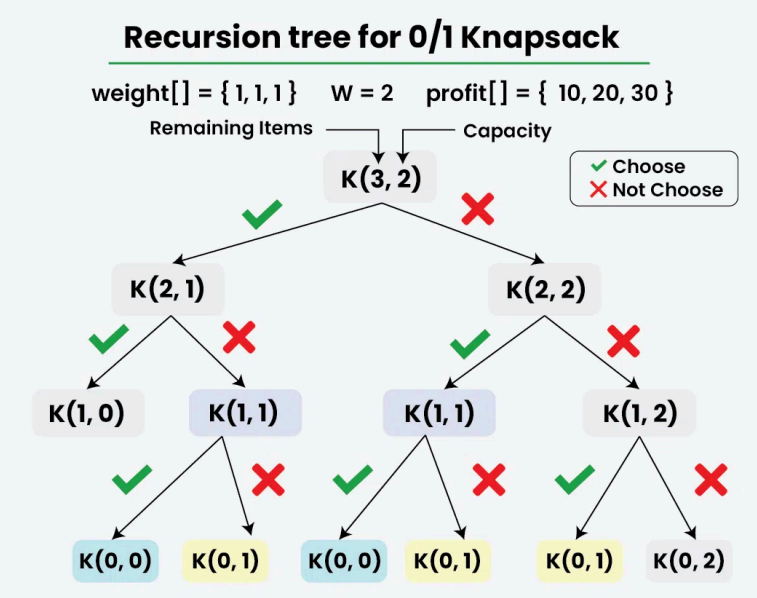
```
220
```

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: The above function using recursion computes the same subproblems again and again. See the following recursion tree, K(1, 1) is being evaluated twice.



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++ Java Python C# JavaScript

```python
# 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
if __name__ == '__main__':
    profit = [60, 100, 120]
    weight = [10, 20, 30]
    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
```

Output

```
220
```

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

Auxiliary Space: O(N * W) + O(N). The use of a 2D array data structure

for 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.*

weight⇢ item⇣/	***0***	***1***	***2***	***3***	***4***	***5***	***6***
0	*0*	*0*	*0*	*0*	*0*	*0*	*0*
1							
2							
3							

- ***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 not put, 2 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], 40+DP[2][3-3]) = max(25, 40) = 40.
  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.

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	0	10	15	40	50	55	65

Below is the implementation of the above approach:

C++ C Java Python C# JavaScript PHP

```cpp
// A dynamic programming based
// solution for 0-1 Knapsack problem
#include <bits/stdc++.h>
using namespace std;

// 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 - 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
```

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:

C++ Java Python C# JavaScript

```cpp
// 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;
}
```

Output

```
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

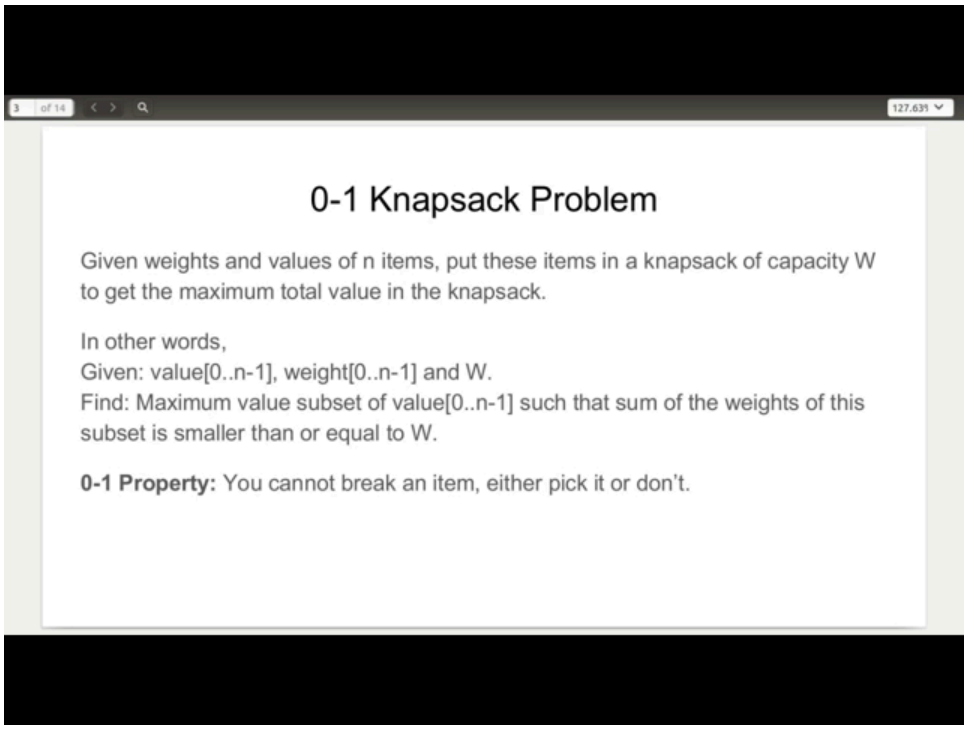
GeeksforGeeks 653

Previous Article

Difference between 0/1 Knapsack problem and Fractional Knapsack problem

Next Article

Printing Items in 0/1 Knapsack



Video | Knapsack Problem - Approach to write the code (Dynamic Programming)

Similar Reads

Difference between 0/1 Knapsack problem and Fractional Knapsack...

What is Knapsack Problem?Suppose you have been given a knapsack or bag with a limited weight capacity, and each item has some weight and...

13 min read

Difference Between Greedy Knapsack and 0/1 Knapsack Algorithms

The 0/1 Knapsack algorithm is a dynamic programming approach where items are either completely included or not at all. It considers all...

9 min read

0/1 Knapsack Problem to print all possible solutions

Given weights and profits of N items, put these items in a knapsack of capacity W. The task is to print all possible solutions to the problem in su...

10 min read

Crossword Puzzle Of The Week #20 (KnapSack Problem)

In this issue of Crossword Puzzle of the Week, we will dive into the topic of the Knapsack Problem. The solution to the crossword puzzle is provided a...

2 min read

GFact | Why doesn't Greedy Algorithm work for 0-1 Knapsack problem?

In this article, we will discuss why the 0-1 knapsack problem cannot be solved by the greedy method, or why the greedy algorithm is not optimal...

2 min read

Fractional Knapsack Problem

Given the weights and profits of N items, in the form of {profit, weight} put these items in a knapsack of capacity W to get the maximum total profit i...

8 min read

A Space Optimized DP solution for 0-1 Knapsack Problem

Given the 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 word...

15+ min read

Extended Knapsack Problem

Given N items, each item having a given weight Ci and a profit value Pi, the task is to maximize the profit by selecting a maximum of K items adding u...

12 min read

Introduction to Knapsack Problem, its Types and How to solve them

The Knapsack problem is an example of the combinational optimization problem. This problem is also commonly known as the "Rucksack...

6 min read

Printing Items in 0/1 Knapsack

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 word...

12 min read

Article Tags : DSA Dynamic Programming knapsack MakeMyTrip +3 More

Practice Tags : MakeMyTrip Snapdeal Visa Zoho +1 More

Kubernetes
Azure
GCP
DevOps Roadmap

Interview Guide
Design Patterns
OOAD
System Design Bootcamp
Interview Questions

School Subjects

Mathematics
Physics
Chemistry
Biology
Social Science
English Grammar

Commerce

Accountancy
Business Studies
Economics
Management
HR Management
Finance
Income Tax

Databases

SQL
MYSQL
PostgreSQL
PL/SQL
MongoDB

Preparation Corner

Company-Wise Recruitment Process
Resume Templates
Aptitude Preparation
Puzzles
Company-Wise Preparation
Companies
Colleges

Competitive Exams

JEE Advanced
UGC NET
UPSC
SSC CGL
SBI PO
SBI Clerk
IBPS PO
IBPS Clerk

More Tutorials

Software Development
Software Testing
Product Management
Project Management
Linux
Excel
All Cheat Sheets
Recent Articles

Free Online Tools

Typing Test
Image Editor
Code Formatters
Code Converters
Currency Converter
Random Number Generator
Random Password Generator

Write & Earn

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

DSA/Placements

Data Structures and Algorithms - Self Paced [Online Course]

Development/Testing

Data Structures & Algorithms in JavaScript - Self Paced Course

Data Structures & Algorithms in Python - Self Paced

C Programming Course Online - Learn C with Data Structures

Complete Interview Preparation

Master Competitive Programming - Complete Beginner to Advanced

Core Computer Science Subject for Interview Preparation

Mastering System Design: From Low-Level to High-Level Solutions

Tech Interview 101 - From DSA to System Design for Working professional [LIVE]

DSA to Development: A complete guide [HYBRID]

Placement Preparation Crash Course [LIVE]

JavaScript Full Course Online | Learn JavaScript with Certification

React JS Course Online - React JS Certification Course

React Native Course Online: Learn React Native Mobile App Development

Complete Django Web Development Course - Basics to Advance

Complete Bootstrap Course For Beginners [Online]

Full Stack Development with React & Node JS - [LIVE]

JAVA Backend Development - [LIVE]

Complete Software Testing Course - Beginner to Advance - [LIVE]

Android Mastery with Kotlin: Beginner to Advanced - [LIVE]

Machine Learning/Data Science

Mastering Generative AI and ChatGPT

Data Analytics Training using Excel, SQL, Python & PowerBI - [LIVE]

Complete Machine Learning & Data Science Program - [LIVE]

Data Science Training Program - [LIVE]

Programming Languages

C Programming Course Online - Learn C with Data Structures

C++ Programming Course Online - Complete Beginner to Advanced

Java Programming Online Course [Complete Beginner to Advanced]

Python Full Course Online - Complete Beginner to Advanced

JavaScript Full Course Online | Learn JavaScript with Certification

Clouds/Devops

DevOps Engineering - Planning to Production

AWS Solutions Architect Certification Live Training Program

Salesforce Certified Administrator Online CourseSalesforce Certified Administrator Online Course

GATE

GATE CS & IT Test Series - 2025

GATE Data Science and Artificial Intelligence Test Series 2025

GATE Computer Science & Information Technology - 2025

GATE Data Science and Artificial Intelligence 2025