**Dynamic Programming**

Dynamic Programming is also used in optimization problems. Like divide-and-conquer method, Dynamic Programming solves problems by combining the solutions of subproblems. Moreover, Dynamic Programming algorithm solves each sub-problem just once and then saves its answer in a table, thereby avoiding the work of re-computing the answer every time.

Two main properties of a problem suggest that the given problem can be solved using Dynamic Programming. These properties are **overlapping sub-problems and optimal substructure**.

## **Overlapping Sub-Problems**

Similar to Divide-and-Conquer approach, Dynamic Programming also combines solutions to sub-problems. It is mainly used where the solution of one sub-problem is needed repeatedly. The computed solutions are stored in a table, so that these don’t have to be re-computed. Hence, this technique is needed where overlapping sub-problem exists.

For example, Binary Search does not have overlapping sub-problem. Whereas recursive program of Fibonacci numbers have many overlapping sub-problems.

## **Optimal Sub-Structure**

A given problem has Optimal Substructure Property, if the optimal solution of the given problem can be obtained using optimal solutions of its sub-problems.

For example, the Shortest Path problem has the following optimal substructure property −

If a node **x** lies in the shortest path from a source node **u** to destination node **v**, then the shortest path from **u** to **v** is the combination of the shortest path from **u** to **x**, and the shortest path from **x** to **v**.

The standard All Pair Shortest Path algorithms like Floyd-Warshall and Bellman-Ford are typical examples of Dynamic Programming.

## **Steps of Dynamic Programming Approach**

Dynamic Programming algorithm is designed using the following four steps −

* Characterize the structure of an optimal solution.
* Recursively define the value of an optimal solution.
* Compute the value of an optimal solution, typically in a bottom-up fashion.
* Construct an optimal solution from the computed information.

## **Applications of Dynamic Programming Approach**

* Matrix Chain Multiplication
* Longest Common Subsequence
* Travelling Salesman Problem

# 0-1 Knapsack

Earlier we have discussed Fractional Knapsack problem using Greedy approach. We have shown that Greedy approach gives an optimal solution for Fractional Knapsack. However, this chapter will cover 0-1 Knapsack problem and its analysis.

In 0-1 Knapsack, items cannot be broken which means the thief should take the item as a whole or should leave it. This is reason behind calling it as 0-1 Knapsack.

Hence, in case of 0-1 Knapsack, the value of ***xi*** can be either ***0*** or ***1***, where other constraints remain the same.

0-1 Knapsack cannot be solved by Greedy approach. Greedy approach does not ensure an optimal solution. In many instances, Greedy approach may give an optimal solution.

The following examples will establish our statement.



### **Example-1**

Let us consider that the capacity of the knapsack is W = 25 and the items are as shown in the following table.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Item** | **A** | **B** | **C** | **D** |
| Profit | 24 | 18 | 18 | 10 |
| Weight | 24 | 10 | 10 | 7 |

Without considering the profit per unit weight (***pi/wi***), if we apply Greedy approach to solve this problem, first item ***A*** will be selected as it will contribute maximum profit among all the elements.

After selecting item ***A***, no more item will be selected. Hence, for this given set of items total profit is ***24***. Whereas, the optimal solution can be achieved by selecting items, ***B*** and C, where the total profit is 18 + 18 = 36.

### **Example-2**

Instead of selecting the items based on the overall benefit, in this example the items are selected based on ratio *pi/wi*. Let us consider that the capacity of the knapsack is *W* = 60 and the items are as shown in the following table.

|  |  |  |  |
| --- | --- | --- | --- |
| Item | A | B | C |
| Price | 100 | 280 | 120 |
| Weight | 10 | 40 | 20 |
| Ratio | 10 | 7 | 6 |

Using the Greedy approach, first item ***A*** is selected. Then, the next item ***B*** is chosen. Hence, the total profit is **100 + 280 = 380**. However, the optimal solution of this instance can be achieved by selecting items, ***B*** and ***C***, where the total profit is **280 + 120 = 400**.

Hence, it can be concluded that Greedy approach may not give an optimal solution.

To solve 0-1 Knapsack, Dynamic Programming approach is required.

### **Problem Statement**

A thief is robbing a store and can carry a maximal weight of ***W*** into his knapsack. There are ***n*** items and weight of **ith** item is ***wi*** and the profit of selecting this item is ***pi***. What items should the thief take?

## **Dynamic-Programming Approach**

Let ***i*** be the highest-numbered item in an optimal solution **S** for **W** dollars. Then ***S' = S - {i}*** is an optimal solution for ***W - wi*** dollars and the value to the solution ***S*** is ***Vi*** plus the value of the sub-problem.

We can express this fact in the following formula: define **c[i, w]** to be the solution for items **1,2, … , i** and the maximum weight **w**.

The algorithm takes the following inputs

* The maximum weight **W**
* The number of items **n**
* The two sequences **v = <v1, v2, …, vn>** and **w = <w1, w2, …, wn>**

**Dynamic-0-1-knapsack (v, w, n, W)**

for w = 0 to W do

c[0, w] = 0

for i = 1 to n do

c[i, 0] = 0

for w = 1 to W do

if wi ≤ w then

if vi + c[i-1, w-wi] then

c[i, w] = vi + c[i-1, w-wi]

else c[i, w] = c[i-1, w]

else

c[i, w] = c[i-1, w]

The set of items to take can be deduced from the table, starting at **c[n, w]** and tracing backwards where the optimal values came from.

If *c[i, w] = c[i-1, w]*, then item ***i*** is not part of the solution, and we continue tracing with **c[i-1, w]**. Otherwise, item ***i*** is part of the solution, and we continue tracing with **c[i-1, w-W]**.

### **Analysis**

This algorithm takes θ(*n*, *w*) times as table *c* has (*n* + 1).(*w* + 1) entries, where each entry requires θ(1) time to compute.

# Dynamic Programming – Subset Sum Problem

**Objective:**Given a set of positive integers, and a value sum S, find out if there exist a subset in array whose sum is equal to given sum S.

Example:

int[] A = { 3, 2, 7, 1}, S = 6

Output: True, subset is (3, 2, 1}

We will first discuss the recursive approach and then we will improve it using [Dynamic Programming](https://algorithms.tutorialhorizon.com/introduction-to-dynamic-programming-fibonacci-series/).

**Recursive Approach:**

For every element in the array has two options, either we will include that element in subset or we don’t include it.

* So if we take example as int[] A = { 3, 2, 7, 1}, S = 6
* If we consider another int array with the same size as A.
* If we include the element in subset we will put 1 in that particular index else put 0.
* So we need to make every possible subsets and check if any of the subset makes the sum as S.
* If we think carefully this problem is quite similar to “[*Generate All Strings of n bits*](https://algorithms.tutorialhorizon.com/generate-all-strings-of-n-bits/)“

**Time Complexity: O(2n).**

**Approach: Dynamic Programming (**[**Bottom-Up**](https://algorithms.tutorialhorizon.com/introduction-to-dynamic-programming-fibonacci-series/)**)**

**Base Cases:**

* If no elements in the set then we can’t make any subset except for 0.
* If sum needed is 0 then by returning the empty subset we can make the subset with sum 0.

**Given** – Set = **arrA[]**, Size = **n**, sum = **S**

* Now for every element in he set we have 2 options, either we include it or exclude it.
* for any ith element-
* If include it => S = S-arrA[i], n=n-1
* If exclude it => S, n=n-1.

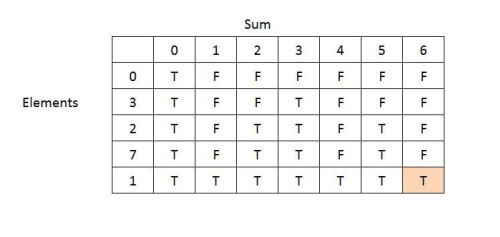
**Recursive Equation:**

**Base Cases:**

SubsetSum (arrA, n, S) = false, if sum > 0 and n == 0 SubsetSum (arrA, n, S)= true, if sum == 0 (return empty set)

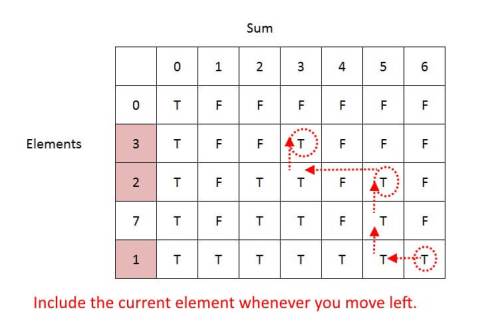
**Rest Cases**

SubsetSum (arrA, n, S) = SubsetSum (arrA, n-1, S) || SubsetSum (arrA, n-1, S-arrA[n-1])

[](https://i2.wp.com/algorithms.tutorialhorizon.com/files/2015/05/Subset-Sum-Problem.jpg)

**How to track the elements.**

* Start from the bottom-right corner and backtrack and check from the True is coming.
* If value in the cell above if false that means current cell has become true after including the current element. So include the current element and check for the sum = sum – current element.

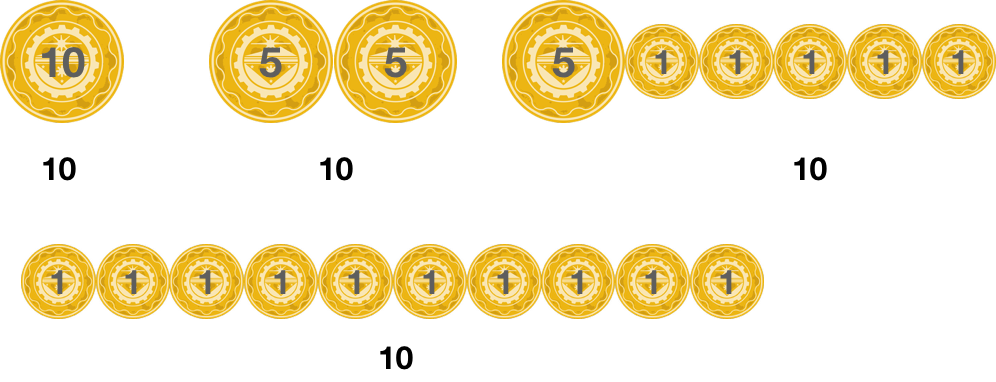
[](https://i1.wp.com/algorithms.tutorialhorizon.com/files/2015/05/Subset-Sum-Problem-Track-Solution.jpg)

**Change making problem/Coin Change Problem | Dynamic Programming**

Coin change problem is the last algorithm we are going to discuss in this section of [dynamic programming](https://www.codesdope.com/course/algorithms-dynamic-programming/). In the coin change problem, we are basically provided with coins with different denominations like 1¢, 5¢ and 10¢. Now, we have to make an amount by using these coins such that a minimum number of coins are used.

Let's take a case of making 10¢ using these coins, we can do it in the following ways:

1. Using 1 coin of 10¢
2. Using two coins of 5¢
3. Using one coin of 5¢ and 5 coins of 1¢
4. Using 10 coins of 1¢



Out of these 4 ways of making 10¢, we can see that the first way of using only one coin of 10¢ requires the least number of coins and thus it is our solution.

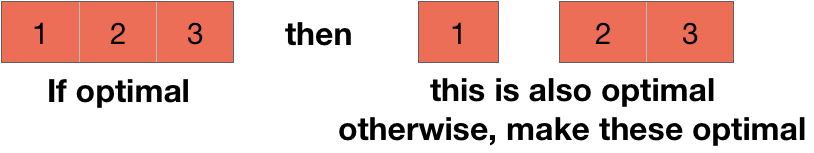
So in a coin change problem, we are provided with different denominations of coins:1=d1<d2<d3<...<dk1=d1<d2<d3<...<dk

d1=1d1=1 ensures that we can make any amount using these coins.

Now, we have to make change for the value nn using these coins and we need to find out the minimum number of coins required to make this change.

## **Approach to Solve the Coin Change Problem**

Like the [rod cutting problem](https://www.codesdope.com/course/algorithms-rod-cutting/), coin change problem also has the property of the optimal substructure i.e., the optimal solution of a problem incorporates the optimal solution to the subproblems. For example, we are making an optimal solution for an amount of 8 by using two values - 5 and 3. So, the optimal solution will be the solution in which 5 and 3 are also optimally made, otherwise, we can reduce the total number of coins of optimizing the values of 5 and 8.



The reason we are checking if the problem has optimal substructure or not because if there is optimal substructure, then the chances are quite high that using dynamic programming will optimize the problem.

Let's say Mn is the minimum number of coins needed to make the change for the value n.

Let's start by picking up the first coin i.e., the coin with the value d1. So, we now need to make the value of n−d1 and Mn−d1 is the minimum number of coins needed for this purpose. So, the total number of coins needed are 1+Mn−d1 (1 coin because we already picked the coin with value d1 and Mn−d1 is the minimum number of coins needed to make the rest of the value).



Similarly, we can pick the second coin first and then attempt to get the optimal solution for the value of n−d2 which will require Mn−d2 coins and thus a total of 1+Mn−d2.

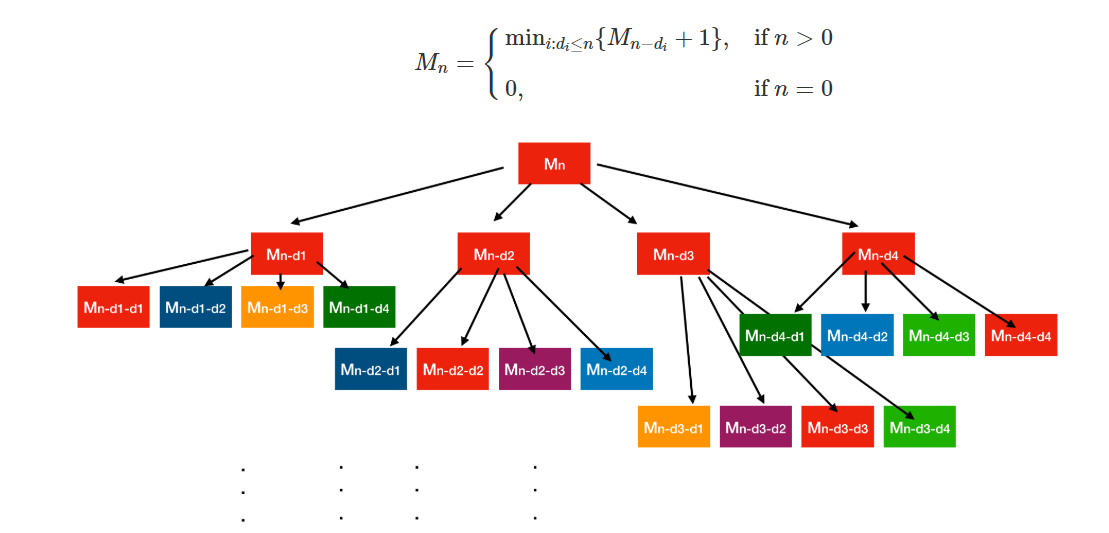
We can repeat the process with all the k coins and then the minimum value of all these will be our answer. i.e., mini:di≤n {Mn−di+1}.



The above process can also be understood in a different way. Suppose, we have picked a coin with value x and we know that this coin is going to be in the solution. So, our next task is to find the minimum number of coins needed to make the change of value n-x i.e., Mn−x. Also, by choosing the coin with value x, we have already increased the total number of coins needed by 1. So, we can write: Mn=1+Mn−x



But the real problem is that we don't know the value of x. So, we will try every coin for the value of x and then we will select the minimum among those.



You can see that there are overlapping subproblems in our solution and thus, we can use dynamic programming to optimize this.

**RELEVANT READING MATERIAL AND REFERENCES:**

**Source Notes:**

* 1. [https://www.tutorialspoint.com/design\_and\_analysis\_of\_algorithms/design\_and\_analysis\_of\_algorithms\_dynamic\_programming.htm#:~:text=Like%20divide%2Dand%2Dconquer%20method,computing%20the%20answer%20every%20time.](https://www.tutorialspoint.com/design_and_analysis_of_algorithms/design_and_analysis_of_algorithms_dynamic_programming.htm" \l ":~:text=Like%20divide%2Dand%2Dconquer%20method,computing%20the%20answer%20every%20time.)
  2. <https://www.tutorialspoint.com/design_and_analysis_of_algorithms/design_and_analysis_of_algorithms_01_knapsack.htm>
  3. <https://algorithms.tutorialhorizon.com/dynamic-programming-subset-sum-problem/>
  4. <https://www.codesdope.com/course/algorithms-coin-change/>

**Lecture Video:**

1. <https://youtu.be/nLmhmB6NzcM>
2. <https://youtu.be/s6FhG--P7z0>
3. <https://youtu.be/Y0ZqKpToTic>

**Online Notes:**

1. <http://vssut.ac.in/lecture_notes/lecture1428551222.pdf>

**Text Book Reading:**

1. Cormen, Leiserson, Rivest, Stein, “*Introduction to Algorithms*”, Prentice Hall of India, 3rd edition 2012. problem, Graph coloring.

**In addition: PPT can be also be given.**