

0/1 Knapsack (medium)

We'll cover the following



- Introduction
- Problem Statement
- Basic Solution
 - Code
 - Time and Space complexity
- Top-down Dynamic Programming with Memoization
 - Code
 - Time and Space complexity
- Bottom-up Dynamic Programming
 - Code
 - Time and Space complexity
 - How can we find the selected items?
- Challenge

Introduction

Given the weights and profits of 'N' items, we are asked to put these items in a knapsack which has a capacity 'C'. The goal is to get the maximum profit out of the items in the knapsack. Each item can only be selected once, as we don't have multiple quantities of any item.

Let's take the example of Merry, who wants to carry some fruits in the knapsack to get maximum profit. Here are the weights and profits of the fruits:

Items: { Apple, Orange, Banana, Melon }

Weights: { 2, 3, 1, 4 }

Profits: { 4, 5, 3, 7 }

Knapsack capacity: 5

Let's try to put various combinations of fruits in the knapsack, such that their total weight is not more than 5:

Apple + Orange (total weight 5) => 9 profit

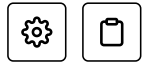
Apple + Banana (total weight 3) => 7 profit

Orange + Banana (total weight 4) => 8 profit

Banana + Melon (total weight 5) => 10 profit

This shows that **Banana + Melon** is the best combination as it gives us the maximum profit and

the total weight does not exceed the capacity.



Problem Statement

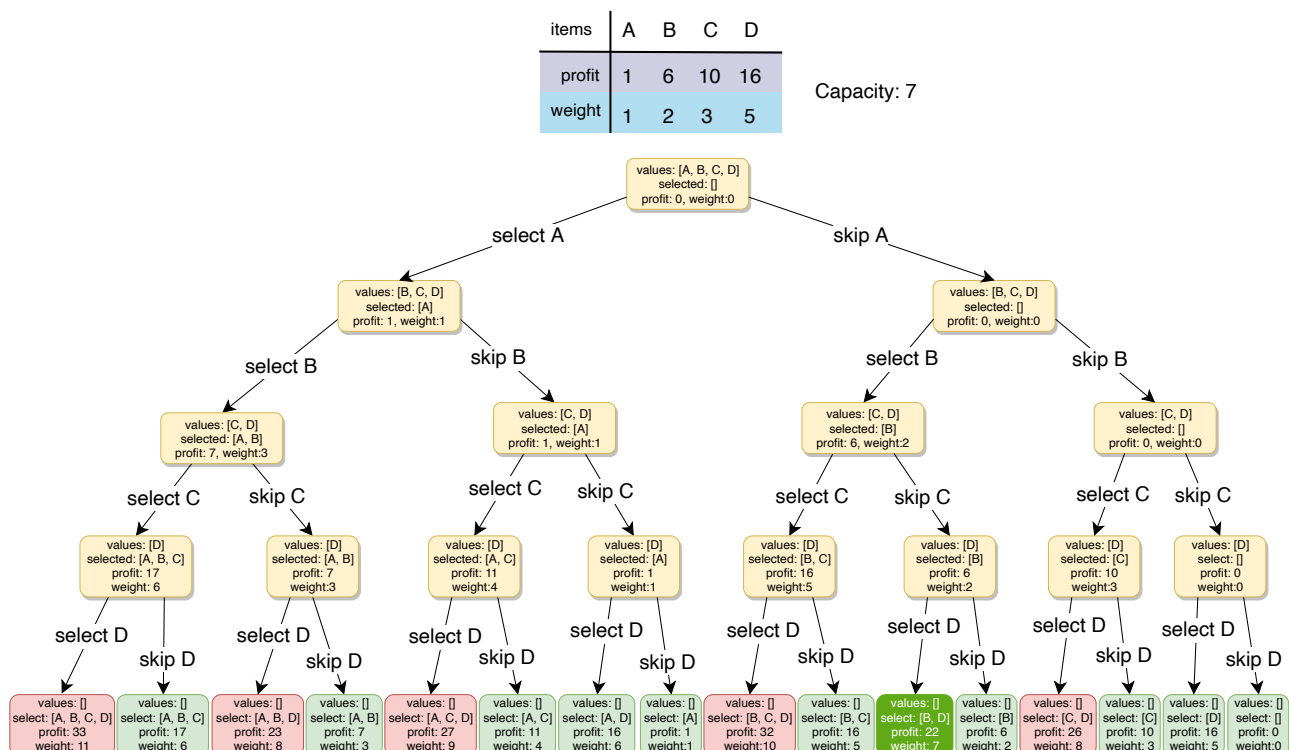
Given two integer arrays to represent weights and profits of 'N' items, we need to find a subset of these items which will give us maximum profit such that their cumulative weight is not more than a given number 'C'. Each item can only be selected once, which means either we put an item in the knapsack or we skip it.

Basic Solution

A basic brute-force solution could be to try all combinations of the given items (as we did above), allowing us to choose the one with maximum profit and a weight that doesn't exceed 'C'. Take the example of four items (A, B, C, and D), as shown in the diagram below. To try all the combinations, our algorithm will look like:

```
1 for each item 'i'
2   create a new set which INCLUDES item 'i' if the total weight does not exceed the capacity
3   recursively process the remaining capacity and items
4   create a new set WITHOUT item 'i', and recursively process the remaining items
5   return the set from the above two sets with higher profit
```

Here is a visual representation of our algorithm:



All green boxes have a total weight that is less than or equal to the capacity (7), and all the red ones have a weight that is more than 7. The best solution we have is with items [B, D] having a total profit of 22 and a total weight of 7.

Code

Here is the code for the brute-force solution:

Java Python3 C++ JS

```
1 def solve_knapsack(profits, weights, capacity):
2     return knapsack_recursive(profits, weights, capacity, 0)
3
4
5 def knapsack_recursive(profits, weights, capacity, currentIndex):
6     # base checks
7     if capacity <= 0 or currentIndex >= len(profits):
8         return 0
9
10    # recursive call after choosing the element at the currentIndex
11    # if the weight of the element at currentIndex exceeds the capacity, we shouldn't process
12    profit1 = 0
13    if weights[currentIndex] <= capacity:
14        profit1 = profits[currentIndex] + knapsack_recursive(
15            profits, weights, capacity - weights[currentIndex], currentIndex + 1)
16
17    # recursive call after excluding the element at the currentIndex
18    profit2 = knapsack_recursive(profits, weights, capacity, currentIndex + 1)
19
20    return max(profit1, profit2)
21
22
23 def main():
24     print(solve_knapsack([1, 6, 10, 16], [1, 2, 3, 5], 7))
25     print(solve_knapsack([1, 6, 10, 16], [1, 2, 3, 5], 6))
26
27
28 main()
```

▶

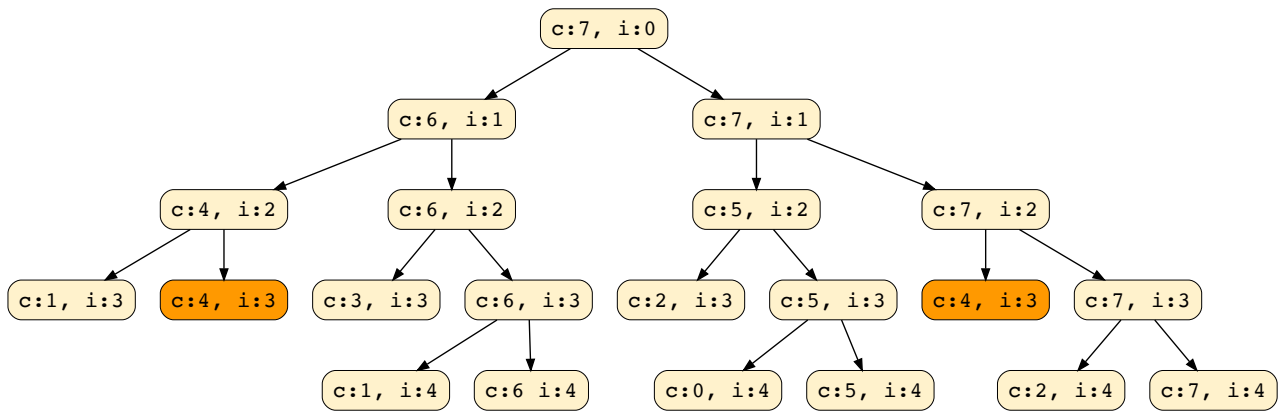
📄 ↶ ⌂

Time and Space complexity

The time complexity of the above algorithm is exponential $O(2^n)$, where 'n' represents the total number of items. This can also be confirmed from the above recursion tree. As we can see, we will have a total of '31' recursive calls – calculated through $(2^n) + (2^n) - 1$, which is asymptotically equivalent to $O(2^n)$.

The space complexity is $O(n)$. This space will be used to store the recursion stack. Since the recursive algorithm works in a depth-first fashion, which means that we can't have more than 'n' recursive calls on the call stack at any time.

Overlapping Sub-problems: Let's visually draw the recursive calls to see if there are any overlapping sub-problems. As we can see, in each recursive call, profits and weights arrays remain constant, and only capacity and currentIndex change. For simplicity, let's denote capacity with 'c' and currentIndex with 'i':



We can clearly see that '**c:4, i=3**' has been called twice. Hence we have an overlapping sub-problems pattern. We can use Memoization (<https://en.wikipedia.org/wiki/Memoization>) to efficiently solve overlapping sub-problems.





Top-down Dynamic Programming with Memoization

Memoization is when we store the results of all the previously solved sub-problems and return the results from memory if we encounter a problem that has already been solved.

Since we have two changing values (capacity and currentIndex) in our recursive function knapsackRecursive() , we can use a two-dimensional array to store the results of all the solved sub-problems. As mentioned above, we need to store results for every sub-array (i.e. for every possible index 'i') and for every possible capacity 'c'.

Code #

Here is the code with memoization (see changes in the highlighted lines):

 Java	 Python3	 C++	 JS
<pre> 1 def solve_knapsack(profits, weights, capacity): 2 # create a two dimensional array for Memoization, each element is initialized to '-1' 3 dp = [[-1 for x in range(capacity+1)] for y in range(len(profits))] 4 return knapsack_recursive(dp, profits, weights, capacity, 0) 5 6 7 def knapsack_recursive(dp, profits, weights, capacity, currentIndex): 8 9 # base checks 10 if capacity <= 0 or currentIndex >= len(profits): 11 return 0 12 13 # if we have already solved a similar problem, return the result from memory 14 if dp[currentIndex][capacity] != -1: 15 return dp[currentIndex][capacity] 16 17 # recursive call after choosing the element at the currentIndex 18 # if the weight of the element at currentIndex exceeds the capacity, we 19 # shouldn't process this 20 profit1 = 0 21 if weights[currentIndex] <= capacity: 22 profit1 = profits[currentIndex] + knapsack_recursive(23 dp, profits, weights, capacity - weights[currentIndex], currentIndex + 1) </pre>			

```

24
25 # recursive call after excluding the element at the currentIndex
26 profit2 = knapsack_recursive(
27     dp, profits, weights, capacity, currentIndex + 1)
28

```



Time and Space complexity

Since our memoization array `dp[profits.length][capacity+1]` stores the results for all subproblems, we can conclude that we will not have more than $N * C$ subproblems (where 'N' is the number of items and 'C' is the knapsack capacity). This means that our time complexity will be $O(N * C)$.

The above algorithm will use $O(N * C)$ space for the memoization array. Other than that we will use $O(N)$ space for the recursion call-stack. So the total space complexity will be $O(N * C + N)$, which is asymptotically equivalent to $O(N * C)$.

Bottom-up Dynamic Programming

Let's try to populate our `dp[][]` array from the above solution by working in a bottom-up fashion. Essentially, we want to find the maximum profit for every sub-array and for every possible capacity. **This means that `dp[i][c]` will represent the maximum knapsack profit for capacity 'c' calculated from the first 'i' items.**

So, for each item at index 'i' ($0 \leq i < \text{items.length}$) and capacity 'c' ($0 \leq c \leq \text{capacity}$), we have two options:

1. Exclude the item at index 'i'. In this case, we will take whatever profit we get from the sub-array excluding this item => `dp[i-1][c]`
2. Include the item at index 'i' if its weight is not more than the capacity. In this case, we include its profit plus whatever profit we get from the remaining capacity and from remaining items => `profit[i] + dp[i-1][c-weight[i]]`

Finally, our optimal solution will be maximum of the above two values:

$$dp[i][c] = \max(dp[i-1][c], \text{profit}[i] + dp[i-1][c - \text{weight}[i]])$$

Let's draw this visually and start with our base case of zero capacity:

			capacity -->							
profit []	weight []	index	0	1	2	3	4	5	6	7
1	1	0	0							
6	2	1	0							
10	3	2	0							
16	5	3	0							



profit []	weight []	index	capacity -->							
			0	1	2	3	4	5	6	7
1	1	0	0	1	1	1	1	1	1	1
6	2	1	0							
10	3	2	0							
16	5	3	0							

Capacity = 1-7, Index = 0, i.e., if we consider the sub-array till index '0', this means we have only one item to put in the knapsack, we will take it if it is not more than the capacity

profit []	weight []	index	capacity -->							
			0	1	2	3	4	5	6	7
1	1	0	0	1	1	1	1	1	1	1
6	2	1	0	1						
10	3	2	0							
16	5	3	0							

Capacity = 1, Index =1, since item at index '1' has weight '2', which is greater than the capacity '1', so we will take the $dp[index-1][capacity]$

profit []	weight []	index	capacity -->							
			0	1	2	3	4	5	6	7
1	1	0	0	1	1	1	1	1	1	1
6	2	1	0	1	6					
10	3	2	0							
16	5	3	0							

Capacity = 2, Index =1, from the formula discussed above: $\max(dp[0][2], profit[1] + dp[0][0])$

			capacity -->							
profit []	weight []	index	0	1	2	3	4	5	6	7
1	1	0	0	1	1	1	1	1	1	1
6	2	1	0	1	6	7				
10	3	2	0							
16	5	3	0							

Capacity = 3, Index =1, from the formula discussed above: $\max(dp[0][3], profit[1] + dp[0][1])$

5 of 23

			capacity -->							
profit []	weight []	index	0	1	2	3	4	5	6	7
1	1	0	0	1	1	1	1	1	1	1
6	2	1	0	1	6	7	7			
10	3	2	0							
16	5	3	0							

Capacity = 4, Index =1, from the formula discussed above: $\max(dp[0][4], profit[1] + dp[0][2])$

6 of 23

			capacity -->							
profit []	weight []	index	0	1	2	3	4	5	6	7
1	1	0	0	1	1	1	1	1	1	1
6	2	1	0	1	6	7	7	7		
10	3	2	0							
16	5	3	0							

Capacity = 5, Index =1, from the formula discussed above: $\max(dp[0][5], profit[1] + dp[0][3])$

7 of 23

			capacity -->							
profit []	weight []	index	0	1	2	3	4	5	6	7
1	1	0	0	1	1	1	1	1	1	1
6	2	1	0	1	6	7	7	7	7	
10	3	2	0							
16	5	3	0							

Capacity = 6, Index =1, from the formula discussed above: $\max(dp[0][6], profit[1] + dp[0][4])$

8 of 23

			capacity -->							
profit []	weight []	index	0	1	2	3	4	5	6	7
1	1	0	0	1	1	1	1	1	1	1
6	2	1	0	1	6	7	7	7	7	7
10	3	2	0							
16	5	3	0							

Capacity = 7, Index =1, from the formula discussed above: $\max(dp[0][7], profit[1] + dp[0][5])$

9 of 23

			capacity -->							
profit []	weight []	index	0	1	2	3	4	5	6	7
1	1	0	0	1	1	1	1	1	1	1
6	2	1	0	1	6	7	7	7	7	7
10	3	2	0	1						
16	5	3	0							

Capacity = 1, Index =2, since item at index '2' has weight '3', which is greater than the capacity '1', so we will take the $dp[index-1][capacity]$

10 of 23

			capacity -->							
profit []	weight []	index	0	1	2	3	4	5	6	7
1	1	0	0	1	1	1	1	1	1	1
6	2	1	0	1	6	7	7	7	7	7
10	3	2	0	1	6					
16	5	3	0							

Capacity = 2, Index =2, since item at index '2' has weight '3', which is greater than the capacity '1', so we will take the $dp[index-1][capacity]$

11 of 23

			capacity -->							
profit []	weight []	index	0	1	2	3	4	5	6	7
1	1	0	0	1	1	1	1	1	1	1
6	2	1	0	1	6	7	7	7	7	7
10	3	2	0	1	6	10				
16	5	3	0							

Capacity = 3, Index =2, from the formula discussed above: $\max(dp[1][3], profit[2] + dp[1][0])$

12 of 23

			capacity -->							
profit []	weight []	index	0	1	2	3	4	5	6	7
1	1	0	0	1	1	1	1	1	1	1
6	2	1	0	1	6	7	7	7	7	7
10	3	2	0	1	6	10	11			
16	5	3	0							

Capacity = 4, Index =2, from the formula discussed above: $\max(dp[1][4], profit[2] + dp[1][1])$

13 of 23

profit []	weight []	index	capacity -->							
			0	1	2	3	4	5	6	7
1	1	0	0	1	1	1	1	1	1	1
6	2	1	0	1	6	7	7	7	7	7
10	3	2	0	1	6	10	11	16		
16	5	3	0							

Capacity = 5, Index =2, from the formula discussed above: $\max(dp[1][5], profit[2] + dp[1][2])$

14 of 23

profit []	weight []	index	capacity -->							
			0	1	2	3	4	5	6	7
1	1	0	0	1	1	1	1	1	1	1
6	2	1	0	1	6	7	7	7	7	7
10	3	2	0	1	6	10	11	16	17	
16	5	3	0							

Capacity = 6, Index =2, from the formula discussed above: $\max(dp[1][6], profit[2] + dp[1][3])$

15 of 23

profit []	weight []	index	capacity -->							
			0	1	2	3	4	5	6	7
1	1	0	0	1	1	1	1	1	1	1
6	2	1	0	1	6	7	7	7	7	7
10	3	2	0	1	6	10	11	16	17	17
16	5	3	0							

Capacity = 7, Index =2, from the formula discussed above: $\max(dp[1][7], profit[2] + dp[1][4])$

16 of 23

			capacity -->							
profit []	weight []	index	0	1	2	3	4	5	6	7
1	1	0	0	1	1	1	1	1	1	1
6	2	1	0	1	6	7	7	7	7	7
10	3	2	0	1	6	10	11	16	17	17
16	5	3	0	1						

Capacity = 1, Index =3, since item at index '3' has weight '5', which is greater than the capacity '1', so we will take the $dp[index-1][capacity]$

17 of 23

			capacity -->							
profit []	weight []	index	0	1	2	3	4	5	6	7
1	1	0	0	1	1	1	1	1	1	1
6	2	1	0	1	6	7	7	7	7	7
10	3	2	0	1	6	10	11	16	17	17
16	5	3	0	1	6					

Capacity = 2, Index =3, since item at index '3' has weight '5', which is greater than the capacity '2', so we will take the $dp[index-1][capacity]$

18 of 23

			capacity -->							
profit []	weight []	index	0	1	2	3	4	5	6	7
1	1	0	0	1	1	1	1	1	1	1
6	2	1	0	1	6	7	7	7	7	7
10	3	2	0	1	6	10	11	16	17	17
16	5	3	0	1	6	10				

Capacity = 3, Index =3, since item at index '3' has weight '5', which is greater than the capacity '3', so we will take the $dp[index-1][capacity]$

19 of 23

			capacity -->							
profit []	weight []	index	0	1	2	3	4	5	6	7
1	1	0	0	1	1	1	1	1	1	1
6	2	1	0	1	6	7	7	7	7	7
10	3	2	0	1	6	10	11	16	17	17
16	5	3	0	1	6	10	11			

Capacity = 4, Index =3, since item at index '3' has weight '5', which is greater than the capacity '4', so we will take the $dp[index-1][capacity]$

20 of 23

			capacity -->							
profit []	weight []	index	0	1	2	3	4	5	6	7
1	1	0	0	1	1	1	1	1	1	1
6	2	1	0	1	6	7	7	7	7	7
10	3	2	0	1	6	10	11	16	17	17
16	5	3	0	1	6	10	11	16		

Capacity = 5, Index =3, from the formula discussed above: $\max(dp[2][5], profit[3] + dp[2][0])$

21 of 23

			capacity -->							
profit []	weight []	index	0	1	2	3	4	5	6	7
1	1	0	0	1	1	1	1	1	1	1
6	2	1	0	1	6	7	7	7	7	7
10	3	2	0	1	6	10	11	16	17	17
16	5	3	0	1	6	10	11	16	17	

Capacity = 6, Index =3, from the formula discussed above: $\max(dp[2][6], profit[3] + dp[2][1])$

22 of 23

			capacity -->							
profit []	weight []	index	0	1	2	3	4	5	6	7
1	1	0	0	1	1	1	1	1	1	1
6	2	1	0	1	6	7	7	7	7	7
10	3	2	0	1	6	10	11	16	17	17
16	5	3	0	1	6	10	11	16	17	22

Capacity = 7, Index =3, from the formula discussed above: $\max(dp[2][7], \text{profit}[3] + dp[2][2])$

23 of 23

— []

Code

Here is the code for our bottom-up dynamic programming approach:

 Java

 Python3

 C++

 JS

```

1 def solve_knapsack(profits, weights, capacity):
2     # basic checks
3     n = len(profits)
4     if capacity <= 0 or n == 0 or len(weights) != n:
5         return 0
6
7     dp = [[0 for x in range(capacity+1)] for y in range(n)]
8
9     # populate the capacity = 0 columns, with '0' capacity we have '0' profit
10    for i in range(0, n):
11        dp[i][0] = 0
12
13    # if we have only one weight, we will take it if it is not more than the capacity
14    for c in range(0, capacity+1):
15        if weights[0] <= c:
16            dp[0][c] = profits[0]
17
18    # process all sub-arrays for all the capacities
19    for i in range(1, n):
20        for c in range(1, capacity+1):
21            profit1, profit2 = 0, 0
22            # include the item, if it is not more than the capacity
23            if weights[i] <= c:
24                profit1 = profits[i] + dp[i - 1][c - weights[i]]
25            # exclude the item
26            profit2 = dp[i - 1][c]
27            # take maximum
28            dp[i][c] = max(profit1, profit2)

```



Time and Space complexity

The above solution has the time and space complexity of $O(N * C)$ where 'N' represents total

The above solution has the time and space complexity of $O(n \times C)$, where 'n' represents total items and 'C' is the maximum capacity.



How can we find the selected items? #

As we know, the final profit is at the bottom-right corner. Therefore, we will start from there to find the items that will be going into the knapsack.

As you remember, at every step we had two options: include an item or skip it. If we skip an item, we take the profit from the remaining items (i.e. from the cell right above it); if we include the item, then we jump to the remaining profit to find more items.

Let's understand this from the above example:

profit	weight	index	capacity -->							
			0	1	2	3	4	5	6	7
1	1	0 (A)	0	1	1	1	1	1	1	1
6	2	1 (B)	0	1	6	7	7	7	7	7
10	3	2 (C)	0	1	6	10	11	16	17	17
16	5	3 (D)	0	1	6	10	11	16	17	22

1. '22' did not come from the top cell (which is 17); hence we must include the item at index '3' (which is item 'D').
2. Subtract the profit of item 'D' from '22' to get the remaining profit '6'. We then jump to profit '6' on the same row.
3. '6' came from the top cell, so we jump to row '2'.
4. Again '6' came from the top cell, so we jump to row '1'.
5. '6' is different than the top cell, so we must include this item (which is item 'B').
6. Subtract the profit of 'B' from '6' to get profit '0'. We then jump to profit '0' on the same row. As soon as we hit zero remaining profit, we can finish our item search.
7. Thus the items going into the knapsack are {B, D}.

Let's write a function to print the set of items included in the knapsack.

Java	Python3	C++	JS
------	---------	-----	----

```
1 from __future__ import print_function
2
3 def solve_knapsack(profits, weights, capacity):
4     # basic checks
5     n = len(profits)
6     if capacity <= 0 or n == 0 or len(weights) != n:
7         return 0
8
9     dp = [[0 for x in range(capacity+1)] for y in range(n)]
10
11     # populate the capacity = 0 columns, with '0' capacity we have '0' profit
12     for i in range(0, n):
13         dp[i][0] = 0
```

```

14
15 # if we have only one weight, we will take it if it is not more than the capacity
16 for c in range(0, capacity+1):
17     if weights[0] <= c:
18         dp[0][c] = profits[0]
19
20 # process all sub-arrays for all the capacities
21 for i in range(1, n):
22     for c in range(1, capacity+1):
23         profit1, profit2 = 0, 0
24         # include the item, if it is not more than the capacity
25         if weights[i] <= c:
26             profit1 = profits[i] + dp[i - 1][c - weights[i]]
27         # exclude the item
28         profit2 = dp[i - 1][c]

```



Challenge

Can we improve our bottom-up DP solution even further? Can you find an algorithm that has $O(C)$ space complexity?

Show Hint

Java

Python3

C++

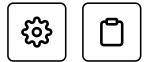
JS

```

1 def solve_knapsack(profits, weights, capacity):
2     # basic checks
3     n = len(profits)
4     if capacity <= 0 or n == 0 or len(weights) != n:
5         return 0
6
7     # we only need one previous row to find the optimal solution, overall we need '2' rows
8     # the above solution is similar to the previous solution, the only difference is that
9     # we use `i % 2` instead of `i` and `(i-1) % 2` instead of `i-1`
10    dp = [[0 for x in range(capacity+1)] for y in range(2)]
11
12    # if we have only one weight, we will take it if it is not more than the capacity
13    for c in range(0, capacity+1):
14        if weights[0] <= c:
15            dp[0][c] = dp[1][c] = profits[0]
16
17    # process all sub-arrays for all the capacities
18    for i in range(1, n):
19        for c in range(0, capacity+1):
20            profit1, profit2 = 0, 0
21            # include the item, if it is not more than the capacity
22            if weights[i] <= c:
23                profit1 = profits[i] + dp[(i - 1) % 2][c - weights[i]]
24            # exclude the item
25            profit2 = dp[(i - 1) % 2][c]
26            # take maximum
27            dp[i % 2][c] = max(profit1, profit2)
28

```





The solution above is similar to the previous solution, the only difference is that we use $i\%2$ instead of i and $(i-1)\%2$ instead of $i-1$. This solution has a space complexity of $O(2 * C) = O(C)$, where 'C' is the maximum capacity of the knapsack.

This space optimization solution can also be implemented using a single array. It is a bit tricky, but the intuition is to use the same array for the previous and the next iteration!

If you see closely, we need two values from the previous iteration: $dp[c]$ and $dp[c - \text{weight}[i]]$

Since our inner loop is iterating over $c:0 \rightarrow \text{capacity}$, let's see how this might affect our two required values:

1. When we access $dp[c]$, it has not been overridden yet for the current iteration, so it should be fine.
2. $dp[c - \text{weight}[i]]$ might be overridden if " $\text{weight}[i] > 0$ ". Therefore we can't use this value for the current iteration.

To solve the second case, we can change our inner loop to process in the reverse direction: $c: \text{capacity} \rightarrow 0$. This will ensure that whenever we change a value in $dp[]$, we will not need it again in the current iteration.

Can you try writing this algorithm?

Java	Python3	C++	JS
<pre>1 def solve_knapsack(profits, weights, capacity): 2 # basic checks 3 n = len(profits) 4 if capacity <= 0 or n == 0 or len(weights) != n: 5 return 0 6 7 dp = [0 for x in range(capacity+1)] 8 9 # if we have only one weight, we will take it if it is not more than the capacity 10 for c in range(0, capacity+1): 11 if weights[0] <= c: 12 dp[c] = profits[0] 13 14 # process all sub-arrays for all the capacities 15 for i in range(1, n): 16 for c in range(capacity, -1, -1): 17 profit1, profit2 = 0, 0 18 # include the item, if it is not more than the capacity 19 if weights[i] <= c: 20 profit1 = profits[i] + dp[c - weights[i]] 21 # exclude the item 22 profit2 = dp[c] 23 # take maximum 24 dp[c] = max(profit1, profit2) 25 26 return dp[capacity] 27 28</pre>			




← Back

Next →

Introduction

Equal Subset Sum Partition (medium)

 **Completed**



Report an
Issue



Ask a Question

(https://discuss.educative.io/tag/01-knapsack-medium__pattern--01-knapsack-dynamic-programming__grokking-the-coding-interview-patterns-for-coding-questions)