



0/1 Knapsack

We'll cover the following



- Introduction
- Problem Statement
- Try it yourself
- Basic Solution
 - Code
- Top-down Dynamic Programming with Memoization
 - Code
- Bottom-up Dynamic Programming
 - Code
 - How to find the selected items?



Introduction

Given the weights and profits of ‘N’ items, we are asked to put these items in a knapsack that has a capacity ‘C’. The goal is to get the maximum profit from 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 Merry’s example, 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 different combinations of fruits in the knapsack, such that their total weight is not more than 5:

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'. Write a function that returns the maximum profit. Each item can only be selected once, which means either we put an item in the knapsack or skip it.

Try it yourself



Try solving this question here:



Java



JS



Python3



C++

```
1 def solve_knapsack(profits, weights, capacity):
2     # TODO: Write your code here
3     return -1
4
5 def main():
6     print(solve_knapsack([1, 6, 10, 16], [1, 2, 3, 5], 7))
7     print(solve_knapsack([1, 6, 10, 16], [1, 2, 3, 5], 6))
8
9
10 main()
```



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:

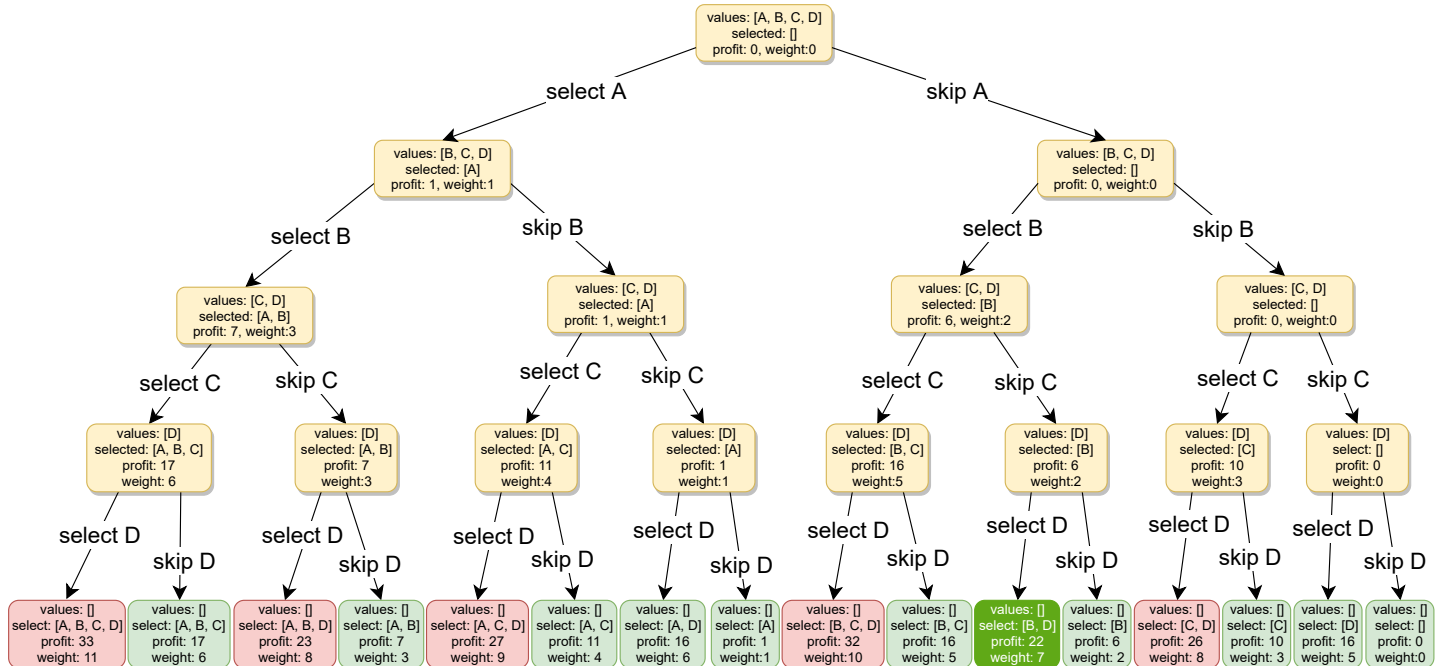
```
1  for each item 'i'
2    create a new set which INCLUDES item 'i' if the total weight does not exceed the capacity
3    recursively process the remaining capacity and items
4    create a new set WITHOUT item 'i', and recursively process the remaining items
5  return the set from the above two sets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

Here is a visual representation of our algorithm:



items	A	B	C	D
profit	1	6	10	16
weight	1	2	3	5

Capacity: 7



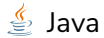
All green boxes have a total weight that is less than or equal to the capacity (7), and all the red ones have a weight that is more than 7. The best solution we have is

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



JS



Python3



C++

```
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,
12    # we shouldn't process this
13    profit1 = 0
14    if weights[currentIndex] <= capacity:
15        profit1 = profits[currentIndex] + knapsack_recursive(
16            profits, weights, capacity - weights[currentIndex], currentIndex + 1)
17
```



```

18 # recursive call after excluding the element at the currentIndex
19 profit2 = knapsack_recursive(profits, weights, capacity, currentIndex + 1)
20
21 return max(profit1, profit2)
22
23
24 def main():
25     print(solve_knapsack([1, 6, 10, 16], [1, 2, 3, 5], 7))
26     print(solve_knapsack([1, 6, 10, 16], [1, 2, 3, 5], 6))
27
28
29 main()

```



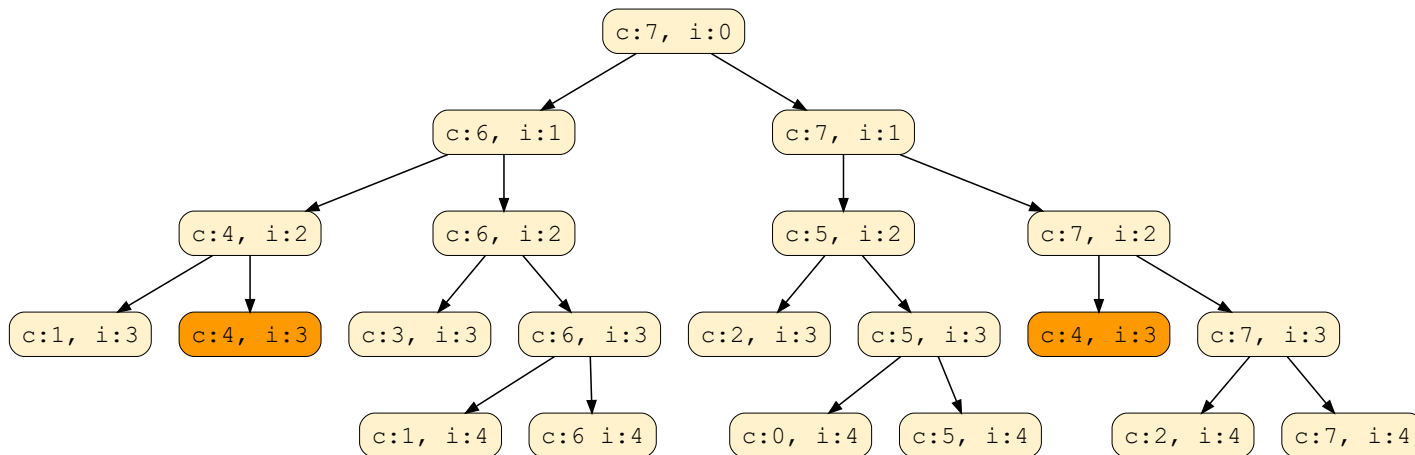
The above algorithm's time complexity is exponential $O(2^n)$, where 'n' represents the total number of items. This can also be confirmed from the above recursion tree. As we can see that we will have a total of '31' recursive calls – calculated through $(2^n) + (2^n) - 1$, which is asymptotically equivalent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

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

can't have more than 1 recursive call on the call stack at any time.



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. As we discussed above, overlapping sub-problems can be solved through Memoization.



Top-down Dynamic Programming with Memoization

We can use memoization to overcome the overlapping sub-problems. To reiterate, 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 the changes in the highlighted lines):



Java



JS



Python3



C++

```
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)
24
```

```

25 # recursive call after excluding the element at the currentIndex
26 profit2 = knapsack_recursive(
27     dp, profits, weights, capacity, currentIndex + 1)
28
29 dp[currentIndex][capacity] = max(profit1, profit2)
30 return dp[currentIndex][capacity]
31

```



What is the time and space complexity of the above solution? Since our memoization array `dp[profits.length][capacity+1]` stores the results for all the 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 be using $O(N * C)$ 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, 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, `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 => `profits[i] + dp[i-1][c-weights[i]]`

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

$$dp[i][c] = \max (dp[i-1][c], profits[i] + dp[i-1][c-weights[i]])$$

Let's visually draw this 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							

With '0' capacity, maximum profit we can have for every subarray is '0'

1 of 23



Code

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

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



Java



JS



Python3



C++

```
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)
29
30     # maximum profit will be at the bottom-right corner.
31     return dp[n - 1][capacity]
```



The above solution has a time and space complexity of $O(N * C)$, where ‘N’ represents total items, and ‘C’ is the maximum capacity.

How to find the selected items?

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

As you remember, at every step, we had two options: include an item or skip it. If we skip an item, then 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 the 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 the 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. So items going into the knapsack are {B, D}.

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



Java



JS



Python3



C++

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



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




Challenge

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



 Show Hint

 Java

 JS

 Python3

 C++

```
1 def solve_knapsack(profits, weights, capacity):
2     # TODO: Write - Your - Code
3     return -1;
4
5
```



Solution



```
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
 29 return dp[(n - 1) % 2][capacity]
 30
 31

The above solution 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 knapsack's maximum capacity.



This space optimization solution can also be implemented using a single array. It is a bit tricky though, 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 anymore in the current iteration.

Can you try writing this algorithm?



 Java

 JS

 Python3

 C++

```
1 def solve_knapsack(profits, weights, capacity):
2     # TODO: Write - Your - Code
3     return -1;
4
```



Solution



```
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]
```

```
12         dp[c] = profits[i]
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
29 def main():
30     print("Total knapsack profit: " +
31           str(solve knapsack([1, 6, 10, 16], [1, 2, 3, 5], 7)))
```

[← Back](#)[Next →](#)

What is Dynamic Programming?

Equal Subset Sum Partition



Mark as Completed



Report an
Issue



Ask a Question

(https://discuss.educative.io/tag/01-knapsack__pattern-1-01-knapsack__grokking-dynamic-programming-patterns-for-coding-interviews)

