

# 0/1 Knapsack

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.

Example:

Items: { Apple, Orange, Banana, Melon }

Weights: { 2, 3, 1, 4 }

Profits: { 4, 5, 3, 7 }

Knapsack weight capacity: 5

Let's try various combinations:

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 (10) and the total weight does not exceed the capacity (5).

Problem Statement #

Given two integer arrays (weights and profits of N items), 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' (Capacity). Each item can only be selected once, which means either we put an item in the knapsack or we skip it (yes/no, 1/0 decision about each item).

# Basic Solution.

Brute-force solution: try all combinations and choose the one with maximum profit and a weight that doesn't exceed "C". Here is the logic:

```
for each item 'i':
    create a new set which INCLUDES item 'i'
    (if the total weight does not exceed the capacity),
    and recursively process the remaining capacity and items
    get profit1
    create a new set WITHOUT item 'i',
    and recursively process the remaining items
    get profit2
return max(profit1,profit2)
```

So, we go through items A,B,C,D in order (for indexes 0,1,2,3)

For each item we branch left (select) or right (skip).

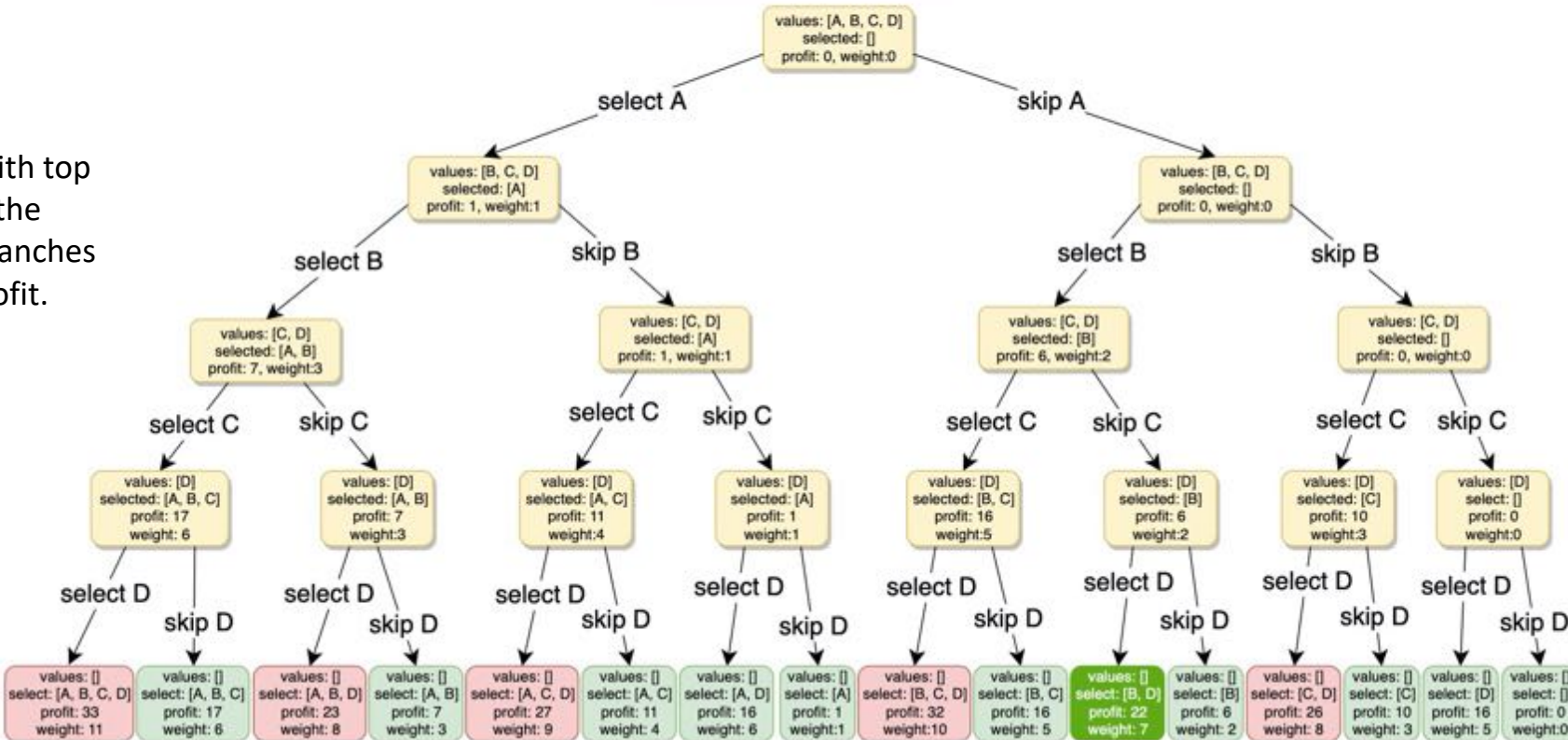
In the end we get combinations which either

- above allowed weight capacity (red),
- or within (green).

And from the green ones we need to select the one with top profit. So we recursively go from bottom up. If one of the branches is red – it will return zero, if we have both branches within capacity – we select the one with maximum profit.

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

Capacity: 7



## 0/1 Knapsack brute-force algorithm

```
# -----  
def solve_knapsack(profits, weights, capacity):  
    return ks_recurs(profits, weights, capacity, 0)  
  
# -----  
def ks_recurs(profits, weights, capacity, idx):  
    if capacity <= 0 or idx >= len(profits):  
        return 0  
  
    # recursive call after choosing the element at the idx  
    # if the weight at idx is > capacity, skip it  
    profit1 = 0  
    if weights[idx] <= capacity:  
        profit1 = profits[idx] + ks_recurs(  
            profits, weights, capacity - weights[idx], idx + 1)  
  
    # recursive call after excluding the element at the idx  
    profit2 = ks_recurs(profits, weights, capacity, idx + 1)  
  
    return max(profit1, profit2)  
  
# -----  
print(solve_knapsack([1, 6, 10, 16], [1, 2, 3, 5], 7))  
print(solve_knapsack([1, 6, 10, 16], [1, 2, 3, 5], 6))
```

Recursive Brute Force:

each item can be selected or not (0 or 1)

Total number of combinations –  $2^N$

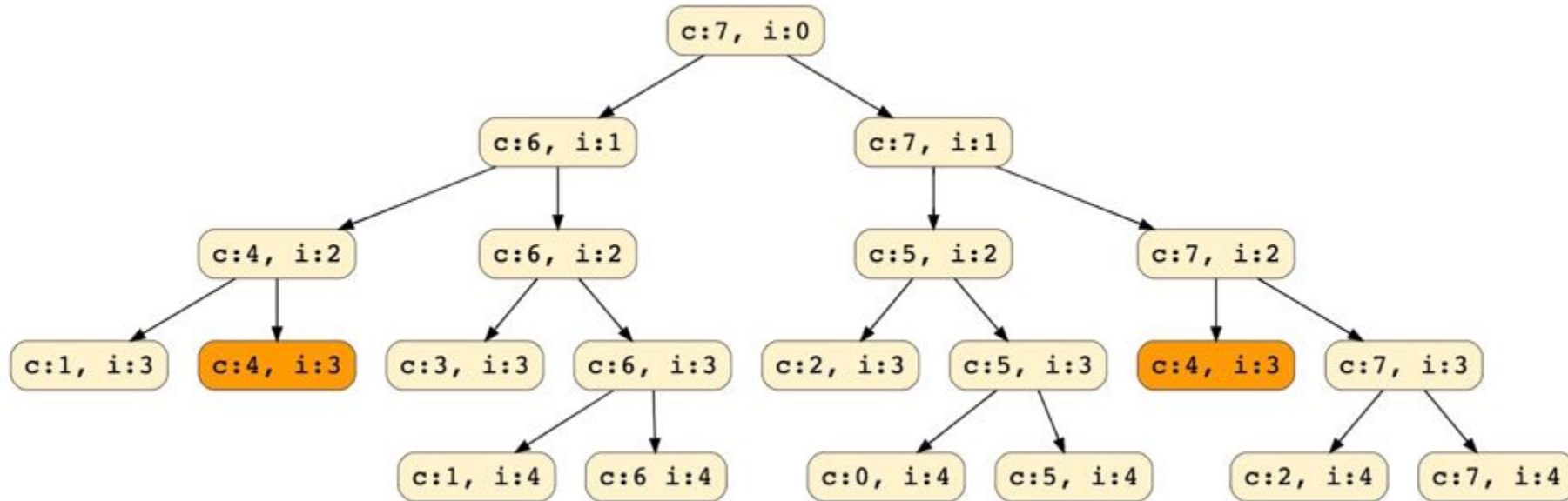
Time Complexity  $O(2^N)$

Space complexity  $O(N)$

We can improve time complexity.

On diagram below we follow execution tree of the code above for the same data as before.

Each node represents a recursive call to `ks_recurs(..., i, c)`, where `i` corresponds to `idx`, `c` – to capacity. As we go deeper, index grows (`0 -> 1 -> 2 -> ...`), whereas remaining capacity shrinks.



Note two orange cells `[c:4, i:3]`. They are the same. We can avoid calculating them twice. After we calculate it once, we can put the value of profit into a two-dimensional array (indexed by index and capacity). So that next time we can avoid calculation – and simply take the value from this array. This trick is called **memoization**.

## 0/1 Knapsack Top-Down Dynamic Programming with Memoization

```
# -----
def solve_knapsack(profits, weights, capacity):
    # Memoization array dp[N_profits][capacity+1] - init with -1
    dp = [[-1 for x in range(capacity+1)] for y in range(len(profits))]
    return ks_recurs(dp, profits, weights, capacity, 0)

# -----
def ks_recurs(dp, profits, weights, capacity, idx):
    if capacity <= 0 or idx >= len(profits):      # base checks
        return 0
    if dp[idx][capacity] != -1: # memoization (return from memory if possible)
        return dp[idx][capacity]

    # recursive call after choosing the element at the idx
    # we skip if weight > capacity
    profit1 = 0
    if weights[idx] <= capacity:
        profit1 = profits[idx] + knapsack_recursive(
            dp, profits, weights, capacity - weights[idx], idx + 1)

    # recursive call after excluding the element at the idx
    profit2 = knapsack_recursive(
        dp, profits, weights, capacity, idx + 1)

    dp[currentIndex][capacity] = max(profit1, profit2)
    return dp[currentIndex][capacity]

# -----
print(solve_knapsack([1, 6, 10, 16], [1, 2, 3, 5], 7))
print(solve_knapsack([1, 6, 10, 16], [1, 2, 3, 5], 6))
```

Top-Down

Dynamic Programming with Memoization.

Basically same as recursive Brute Force.

We just added memoization array.

It stores  $\sim N \times C$  values. So:

Time Complexity  $O(N \times C)$

Space complexity  $O(N \times C)$

How can we optimize the solution even more?  
 Let's try to change direction from Top-Down to Bottom-Up.  
 We want to populate our memoization cache array `dp[i][c]` by working in a bottom-up fashion.

`dp[i][c]` will represent the max knapsack profit for capacity "c" calculated from the first "i" items.

- Similar to how we were doing before, for each combination `[i][c]` we have two options:
- Exclude the item at index "i" – and take whatever profit we get from the sub-array excluding this item:  
 => `dp[i-1][c]`
  - Include the item at index "i" (of course 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 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] , profit[i] + dp[i-1][c-weight[i]] )`

So, we will loop through index and capacity like this:

```

for i in range(1, n):
    for c in range(1, capacity+1):
        ...
    
```

The `dp[i][c]` values (calculated using above logic) are shown on the right. The max value is 22, and it corresponds to max values of `i=3, c=7`.

The actual code is show on the next page.

			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], profit[3] + dp[2][2] )`

## 0/1 Knapsack Bottom-Up populating of dp[i][c]

```
def solve_knapsack(profits, weights, capacity):
    n = len(profits)
    if capacity <= 0 or n == 0 or len(weights) != n:
        return 0
    dp = [[0 for x in range(capacity+1)] for y in range(n)]
    for i in range(0, n):
        dp[i][0] = 0      # with 0 capacity we have 0 profits

    # if we have only one weight, we will take it
    for c in range(0, capacity+1):
        if weights[0] <= c:
            dp[0][c] = profits[0]

    # process all sub-arrays for all the capacities
    for i in range(1, n):
        for c in range(1, capacity+1):
            profit1, profit2 = 0, 0
            # include the item, if it is not more than the capacity
            if weights[i] <= c:
                profit1 = profits[i] + dp[i - 1][c - weights[i]]
            # exclude the item
            profit2 = dp[i - 1][c]
            # take maximum
            dp[i][c] = max(profit1, profit2)

    # maximum profit will be at the bottom-right corner.
    return dp[n - 1][capacity]

# -----
print(solve_knapsack([1, 6, 10, 16], [1, 2, 3, 5], 5))
print(solve_knapsack([1, 6, 10, 16], [1, 2, 3, 5], 6))
print(solve_knapsack([1, 6, 10, 16], [1, 2, 3, 5], 7))
# -----
```

Bottom-Up populating of d[i][c]

Time Complexity  $O(N \cdot C)$

Space complexity  $O(N \cdot C)$



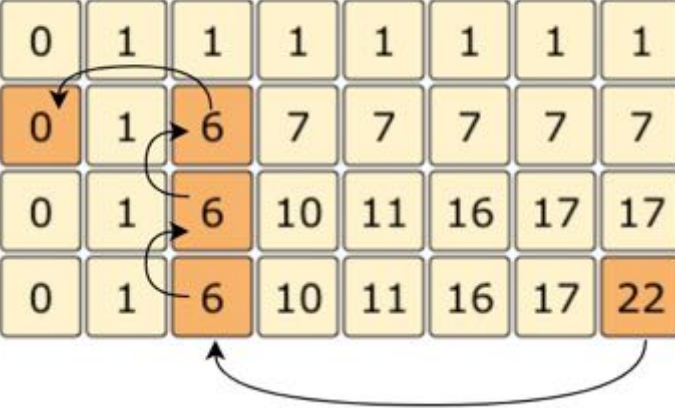
# How can we find the selected items ?

(if we using the last code (Bottom-Up approach))

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:

			capacity -->							
profit	weight	index	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



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

'22' did not come from the top cell (which is 17);  
hence we must include the item at index '3' (which is item 'D').

Subtract the profit of item 'D' from '22' to get the remaining profit '6'.  
We then jump to profit '6' on the same row.

'6' came from the top cell, so we jump to row '2'.  
Again '6' came from the top cell, so we jump to row '1'.  
'6' is different than the top cell, so we must include this item (which is item 'B').

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.  
Thus the items going into the knapsack are {B, D}.

```
def print_selected(dp, weights, profits, capacity):
    print("Selected weights are: ", end='')
    n = len(weights)
    totalProfit = dp[n-1][capacity]
    for i in range(n-1, 0, -1):
        if totalProfit != dp[i-1][capacity]:
            print(str(weights[i]) + " ", end='')
            capacity -= weights[i]
            totalProfit -= profits[i]

    if totalProfit != 0:
        print(str(weights[0]) + " ", end='')
    print()
```



# 0/1 Knapsack Bottom-Up populating only two rows of dp[][]

```
def solve_knapsack(profits, weights, capacity):
    n = len(profits)
    if capacity <= 0 or n == 0 or len(weights) != n:
        return 0

    # we only need one previous row, so we need only 2 rows.
    # we use "i % 2" instead of "i" and "(i-1) % 2" instead of "i-1"
    dp = [[0 for x in range(capacity+1)] for y in range(2)]

    # if we have only one weight, we will take it
    for c in range(0, capacity+1):
        if weights[0] <= c:
            dp[0][c] = dp[1][c] = profits[0]

    # process all sub-arrays for all the capacities
    for i in range(1, n):
        for c in range(0, capacity+1):
            profit1, profit2 = 0, 0
            # include the item, if it is not more than the capacity
            if weights[i] <= c:
                profit1 = profits[i] + dp[(i - 1) % 2][c - weights[i]]
            # exclude the item
            profit2 = dp[(i - 1) % 2][c]
            # take maximum
            dp[i % 2][c] = max(profit1, profit2)

    return dp[(n - 1) % 2][capacity]

# -----
print("Total knapsack profit: " +
      str(solve_knapsack([1, 6, 10, 16], [1, 2, 3, 5], 7)))
print("Total knapsack profit: " +
      str(solve_knapsack([1, 6, 10, 16], [1, 2, 3, 5], 6)))
```

We can improve the space complexity – make it  $O(C)$ .

The time complexity will still be  $O(N \cdot C)$

Hint - we only need one previous row to find the optimal solution!  
So we don't need to keep the whole array  $d[][]$ .

Bottom-Up populating only of  $d[i][c]$

Time Complexity  $O(N \cdot C)$

Space complexity  $O(C)$

Note - we can simplify this solution even further by using only one array.  
Idea is to use the same array for the previous and the next iteration!

We need two values from the previous iteration:  
 $dp[c]$  and  $dp[c - \text{weight}[i]]$

Our inner loop is iterating over  $c$  (**0 .. capacity**).

When we access  $dp[c]$ , it has not been overridden yet for the current iteration, so it should be fine.

But  $dp[c - \text{weight}[i]]$  might be overridden if " $\text{weight}[i] > 0$ ".  
Therefore we can't use this value for the current iteration.  
But this problem can be solved easily by reversing the order of the inner loop over  $c$ : (**capacity ... 0**)  
This will ensure that whenever we change a value in  $dp[]$ , we will not need it again in the current iteration.

## 0/1 Knapsack Bottom-Up using only one array

```
def solve_knapsack(profits, weights, capacity):
    n = len(profits)
    if capacity <= 0 or n == 0 or len(weights) != n:
        return 0

    dp = [0 for x in range(capacity+1)]

    for c in range(0, capacity+1):
        if weights[0] <= c:
            dp[c] = profits[0]

    # process all sub-arrays for all the capacities
    for i in range(1, n):
        for c in range(capacity, -1, -1):
            profit1, profit2 = 0, 0
            # include the item, if it is not more than the capacity
            if weights[i] <= c:
                profit1 = profits[i] + dp[c - weights[i]]
            # exclude the item
            profit2 = dp[c]
            # take maximum
            dp[c] = max(profit1, profit2)

    return dp[capacity]

# -----
print("Total knapsack profit: " +
      str(solve_knapsack([1, 6, 10, 16], [1, 2, 3, 5], 7)))
print("Total knapsack profit: " +
      str(solve_knapsack([1, 6, 10, 16], [1, 2, 3, 5], 6)))
```

Final optimized solution using only one array.

Time Complexity  $O(N \cdot C)$

Space complexity  $O(C)$