Unbounded Knapsack

We'll cover the following

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 from the items in the knapsack. The only difference between the 0/1 Knapsack (https://www.educative.io/collection/page/5668639101419520/5633779737559040/5666387129270272/) problem and this problem is that we are allowed to use an unlimited quantity of an 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, Melon }
Weights: { 1, 2, 3 }
Profits: { 15, 20, 50 }
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.

5 Apples (total weight 5) => 75 profit
1 Apple + 2 Oranges (total weight 5) => 55 profit
2 Apples + 1 Melon (total weight 5) => 80 profit
1 Orange + 1 Melon (total weight 5) => 70 profit

This shows that **2 apples** + **1 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’. We can assume an infinite supply of item quantities; therefore, each item can be selected multiple times.

Basic Solution

A basic brute-force solution could be to try all combinations of the given items to choose the one with maximum profit and a weight that doesn’t exceed ‘C’. This is what our algorithm will look like:

```
for each item 'i'
  create a new set which includes one quantity of item 'i' if it does not exceed the capac
     recursively call to process all items
  create a new set without item 'i', and recursively process the remaining items
return the set from the above two sets with higher profit
```

The only difference between the 0/1 Knapsack (https://www.educative.io/collection/page/5668639101419520/5633779737559040/5666387129270272/) problem and this one is that, after including the item, we recursively call to process all the items (including the current item). In 0/1 Knapsack, however, we recursively call to process the remaining items.

Code

Here is the code for the brute-force solution:

Java | JS | Python3 | C++

```
def solve_knapsack(profits, weights, capacity):
  return solve_knapsack_recursive(profits, weights, capacity, 0)


def solve_knapsack_recursive(profits, weights, capacity, currentIndex):
  n = len(profits)
  # base checks
  if capacity <= 0 or n == 0 or len(weights) != n or currentIndex >= n:
    return 0

  # recursive call after choosing the items at the currentIndex, note that we recursive ca
  # items as we did not increment currentIndex
  profit1 = 0
  if weights[currentIndex] <= capacity:
    profit1 = profits[currentIndex] + solve_knapsack_recursive(
      profits, weights, capacity - weights[currentIndex], currentIndex)

  # recursive call after excluding the element at the currentIndex
  profit2 = solve_knapsack_recursive(
    profits, weights, capacity, currentIndex + 1)

  return max(profit1, profit2)


def main():
  print(solve_knapsack([15, 50, 60, 90], [1, 3, 4, 5], 8))
  print(solve_knapsack([15, 50, 60, 90], [1, 3, 4, 5], 6))
```

```
28
29
30  main()
```

The time complexity of the above algorithm is exponential $O(2^{N+C})$, where 'N' represents the total number of items. The space complexity will be $O(N + C)$ to store the recursion stack.

Let's try to find a better solution.

Top-down Dynamic Programming with Memoization

Once again, we can use memoization to overcome the overlapping sub-problems.

We will be using a two-dimensional array to store the results of solved sub-problems. As mentioned above, we need to store results for every sub-array and for every possible capacity. Here is the code:

Java | JS | Python3 | C++

```python
def solve_knapsack(profits, weights, capacity):
  dp = [[-1 for _ in range(capacity+1)] for _ in range(len(profits))]
  return solve_knapsack_recursive(dp, profits, weights, capacity, 0)


def solve_knapsack_recursive(dp, profits, weights, capacity, currentIndex):
  n = len(profits)
  # base checks
  if capacity <= 0 or n == 0 or len(weights) != n or currentIndex >= n:
    return 0

  # check if we have not already processed a similar sub-problem
  if dp[currentIndex][capacity] == -1:
    # recursive call after choosing the items at the currentIndex, note that we
    # recursive call on all items as we did not increment currentIndex
    profit1 = 0
    if weights[currentIndex] <= capacity:
      profit1 = profits[currentIndex] + solve_knapsack_recursive(
        dp, profits, weights, capacity - weights[currentIndex], currentIndex)

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

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

  return dp[currentIndex][capacity]


def main():
  print(solve_knapsack([15, 50, 60, 90], [1, 3, 4, 5], 8))
  print(solve_knapsack([15, 50, 60, 90], [1, 3, 4, 5], 6))


main()
```

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, what we want to achieve is: "Find the maximum profit for every sub-array and for every possible capacity".

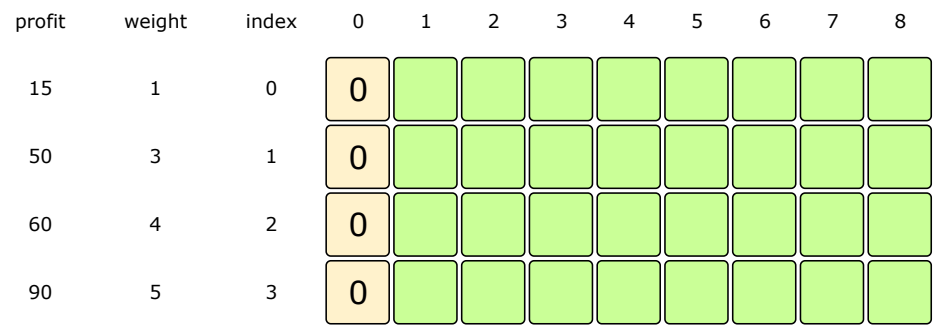
So for every possible capacity 'c' (0 <= c <= capacity), we have two options:

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

Finally, we have to take the maximum of the above two values:

```
    dp[index][c] = max (dp[index-1][c], profit[index] + dp[index][c-weight[index
]])
```

Let's start with our base case of zero capacity:



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

profit	weight	index	0	1	2	3	4	5	6	7	8
15	1	0	0	15							
50	3	1	0								
60	4	2	0								
90	5	3	0								

Capacity = 1, Index = 0, i.e., if we consider the sub-array till index '0', maximum profit will be '1'5, as we can fit the item with weight '1' in the knapsack

profit	weight	index	0	1	2	3	4	5	6	7	8
15	1	0	0	15	30						
50	3	1	0								
60	4	2	0								
90	5	3	0								

Capacity = 2, Index = 0, => profits[Index] + dp[Index][1], we are not considering dp[index-1][Capacity] as Index is not bigger than '0'

profit	weight	index	0	1	2	3	4	5	6	7	8
15	1	0	0	15	30	45					
50	3	1	0								
60	4	2	0								
90	5	3	0								

Capacity = 3, Index = 0, => profits[Index] + dp[Index][1]

profit	weight	index	0	1	2	3	4	5	6	7	8
15	1	0	0	15	30	45	60	75	90	105	120
50	3	1	0								
60	4	2	0								
90	5	3	0								

Capacity = 4-8, Index = 0, => profits[Index] + dp[Index][1]

profit	weight	index	0	1	2	3	4	5	6	7	8
15	1	0	0	15	30	45	60	75	90	105	120
50	3	1	0	15							
60	4	2	0								
90	5	3	0								

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

profit	weight	index	0	1	2	3	4	5	6	7	8
15	1	0	0	15	30	45	60	75	90	105	120
50	3	1	0	15	30						
60	4	2	0								
90	5	3	0								

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

profit	weight	index	0	1	2	3	4	5	6	7	8
15	1	0	0	15	30	45	60	75	90	105	120
50	3	1	0	15	30	50					
60	4	2	0								
90	5	3	0								

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

profit	weight	index	0	1	2	3	4	5	6	7	8
15	1	0	0	15	30	45	60	75	90	105	120
50	3	1	0	15	30	50	65				
60	4	2	0								
90	5	3	0								

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

profit	weight	index	0	1	2	3	4	5	6	7	8
15	1	0	0	15	30	45	60	75	90	105	120
50	3	1	0	15	30	50	65	80			
60	4	2	0								
90	5	3	0								

Capacity = 5, Index =1, from the formula discussed above: max(dp[0][5], profit[1] + dp[1][2])

profit	weight	index	0	1	2	3	4	5	6	7	8
15	1	0	0	15	30	45	60	75	90	105	120
50	3	1	0	15	30	50	65	80	100		
60	4	2	0								
90	5	3	0								

Capacity = 6, Index =1, from the formula discussed above: max(dp[0][6], profit[1] + dp[1][3])

profit	weight	index	0	1	2	3	4	5	6	7	8
15	1	0	0	15	30	45	60	75	90	105	120
50	3	1	0	15	30	50	65	80	100	115	
60	4	2	0								
90	5	3	0								

Capacity = 7, Index =1, from the formula discussed above: max(dp[0][7], profit[1] + dp[1][4])

profit	weight	index	0	1	2	3	4	5	6	7	8
15	1	0	0	15	30	45	60	75	90	105	120
50	3	1	0	15	30	50	65	80	100	115	130
60	4	2	0								
90	5	3	0								

Capacity = 8, Index =1, from the formula discussed above: max(dp[0][8], profit[1] + dp[1][5])

profit	weight	index	0	1	2	3	4	5	6	7	8
15	1	0	0	15	30	45	60	75	90	105	120
50	3	1	0	15	30	50	65	80	100	115	130
60	4	2	0	15	30	50					
90	5	3	0								

Capacity = 1-3, Index =2, since item at index '2' has weight '4', which is greater than the capacity, so we will take the dp[index-1][capacity]

profit	weight	index	0	1	2	3	4	5	6	7	8
15	1	0	0	15	30	45	60	75	90	105	120
50	3	1	0	15	30	50	65	80	100	115	130
60	4	2	0	15	30	50	65				
90	5	3	0								

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

profit	weight	index	0	1	2	3	4	5	6	7	8
15	1	0	0	15	30	45	60	75	90	105	120
50	3	1	0	15	30	50	65	80	100	115	130
60	4	2	0	15	30	50	65	80			
90	5	3	0								

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

profit	weight	index	0	1	2	3	4	5	6	7	8
15	1	0	0	15	30	45	60	75	90	105	120
50	3	1	0	15	30	50	65	80	100	115	130
60	4	2	0	15	30	50	65	80	100		
90	5	3	0								

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

profit	weight	index	0	1	2	3	4	5	6	7	8
15	1	0	0	15	30	45	60	75	90	105	120
50	3	1	0	15	30	50	65	80	100	115	130
60	4	2	0	15	30	50	65	80	100	115	
90	5	3	0								

Capacity = 7, Index =2, from the formula discussed above: max(dp[1][7], profit[2] + dp[2][3])

profit	weight	index	0	1	2	3	4	5	6	7	8
15	1	0	0	15	30	45	60	75	90	105	120
50	3	1	0	15	30	50	65	80	100	115	130
60	4	2	0	15	30	50	65	80	100	115	130
90	5	3	0								

Capacity = 8, Index =2, from the formula discussed above: max(dp[1][8], profit[2] + dp[2][4])

profit	weight	index	0	1	2	3	4	5	6	7	8
15	1	0	0	15	30	45	60	75	90	105	120
50	3	1	0	15	30	50	65	80	100	115	130
60	4	2	0	15	30	50	65	80	100	115	130
90	5	3	0	15	30	50	65				

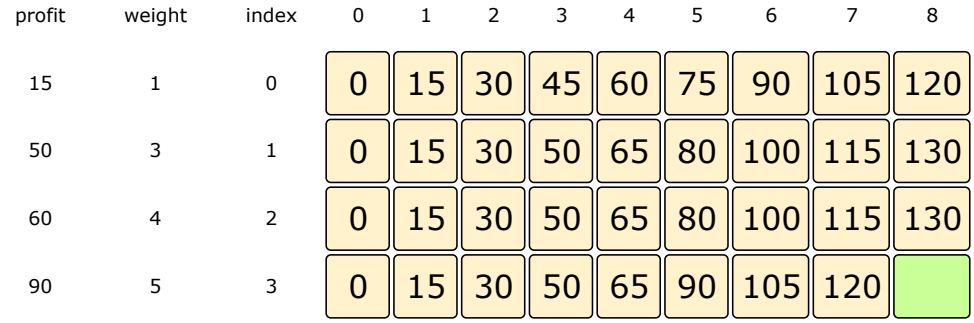
Capacity = 1-4, Index =3, since item at index '3' has weight '5', which is greater than the capacity, so we will take the dp[index-1][capacity]

profit	weight	index	0	1	2	3	4	5	6	7	8
15	1	0	0	15	30	45	60	75	90	105	120
50	3	1	0	15	30	50	65	80	100	115	130
60	4	2	0	15	30	50	65	80	100	115	130
90	5	3	0	15	30	50	65	90			

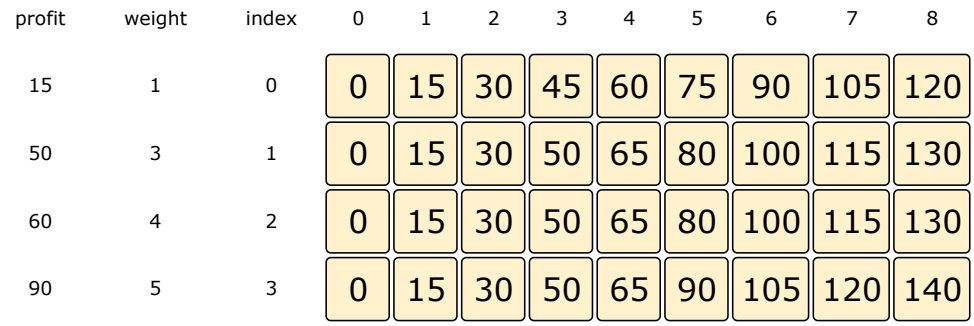
Capacity = 5, Index =3, from the formula discussed above: max(dp[2][5], profit[3] + dp[3][0])

profit	weight	index	0	1	2	3	4	5	6	7	8
15	1	0	0	15	30	45	60	75	90	105	120
50	3	1	0	15	30	50	65	80	100	115	130
60	4	2	0	15	30	50	65	80	100	115	130
90	5	3	0	15	30	50	65	90	105		

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



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



Capacity = 8, Index =3, from the formula discussed above: max(dp[2][8], profit[3] + dp[3][3])

Code

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

Java | JS | Python3 | C++

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

  dp = [[-1 for _ in range(capacity+1)] for _ in range(len(profits))]

  # populate the capacity=0 columns
  for i in range(n):
    dp[i][0] = 0

  # process all sub-arrays for all capacities
  for i in range(n):
    for c in range(1, capacity+1):
      profit1, profit2 = 0, 0
      if weights[i] <= c:
        profit1 = profits[i] + dp[i][c - weights[i]]
      if i > 0:
```

```
            profit2 = dp[i - 1][c]
        dp[i][c] = profit1 if profit1 > profit2 else profit2

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


def main():
    print(solve_knapsack([15, 50, 60, 90], [1, 3, 4, 5], 8))
    print(solve_knapsack([15, 50, 60, 90], [1, 3, 4, 5], 6))


main()
```

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

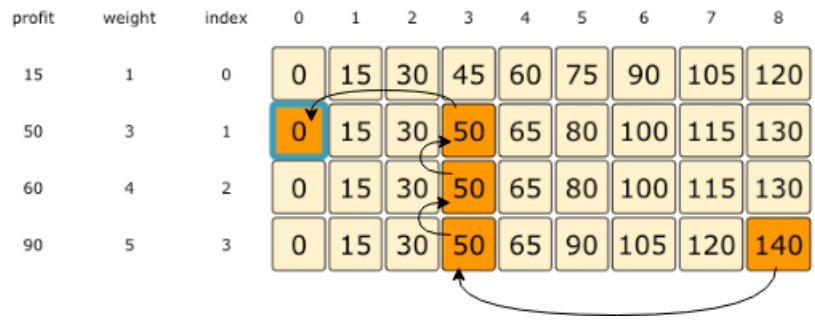
Find the selected items

As we know, the final profit is at the right-bottom corner; hence we will start from there to find the items that will be going to 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 cell right above it; if we include the item, then we jump to the remaining profit to find more items.

Let's assume the four items are identified as {A, B, C, and D}, and use the above example to better understand this:

1. '140' did not come from the top cell (which is 130); hence we must include the item at index '3', which is 'D'.
2. Subtract the profit of 'D' from '140' to get the remaining profit '50'. We then jump to profit '50' on the same row.
3. '50' came from the top cell, so we jump to row '2'.
4. Again, '50' came from the top cell, so we jump to row '1'.
5. '50' is different than the top cell, so we must include this item, which is 'B'.
6. Subtract the profit of 'B' from '50' to get the remaining 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}.



Back

Target Sum

Next

Rod Cutting

Completed