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**[#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/qV6RXWME4D3#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.

### Top-down Dynamic Programming with Memoization [#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/qV6RXWME4D3#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:

class Knapsack {

  public int solveKnapsack(int[] profits, int[] weights, int capacity) {

    Integer[][] dp = new Integer[profits.length][capacity + 1];

    return this.knapsackRecursive(dp, profits, weights, capacity, 0);

  }

  private int knapsackRecursive(Integer[][] dp, int[] profits, int[] weights, int capacity,

      int currentIndex) {

    // base checks

    if (capacity <= 0 || profits.length == 0 || weights.length != profits.length ||

        currentIndex >= profits.length)

      return 0;

    // check if we have not already processed a similar sub-problem

    if(dp[currentIndex][capacity] == null) {

      // recursive call after choosing the items at the currentIndex, note that we recursive call on all

      // items as we did not increment currentIndex

      int profit1 = 0;

      if( weights[currentIndex] <= capacity )

          profit1 = profits[currentIndex] + knapsackRecursive(dp, profits, weights,

                  capacity - weights[currentIndex], currentIndex);

      // recursive call after excluding the element at the currentIndex

      int profit2 = knapsackRecursive(dp, profits, weights, capacity, currentIndex + 1);

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

    }

    return dp[currentIndex][capacity];

  }

  public static void main(String[] args) {

    Knapsack ks = new Knapsack();

    int[] profits = {15, 50, 60, 90};

    int[] weights = {1, 3, 4, 5};

    System.out.println(ks.solveKnapsack(profits, weights, 8));

    System.out.println(ks.solveKnapsack(profits, weights, 6));

  }

}

**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*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)*O*(*N*∗*C*).

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

### Bottom-up Dynamic Programming [#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/qV6RXWME4D3#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”.

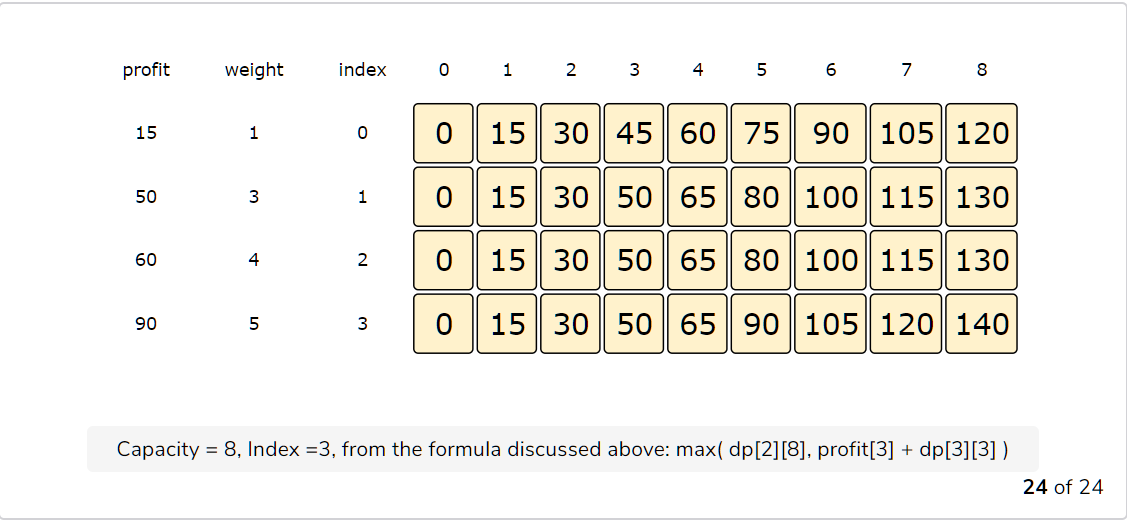
So for every possible capacity ‘c’ (0 <= c <= capacity), we have two options:

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]]

[[1]](#footnote-1)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:



  public int solveKnapsack(int[] profits, int[] weights, int capacity) {

    // base checks

    if (capacity <= 0 || profits.length == 0 || weights.length != profits.length)

      return 0;

    int n = profits.length;

    int[][] dp = new int[n][capacity + 1];

    // populate the capacity=0 columns

    for(int i=0; i < n; i++)

      dp[i][0] = 0;

    // process all sub-arrays for all capacities

    for(int i=0; i < n; i++) {

      for(int c=1; c <= capacity; c++) {

        int profit1=0, profit2=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 > profit2 ? profit1 : profit2;

      }

    }

    // maximum profit will be in the bottom-right corner.

    return dp[n-1][capacity];

  }

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

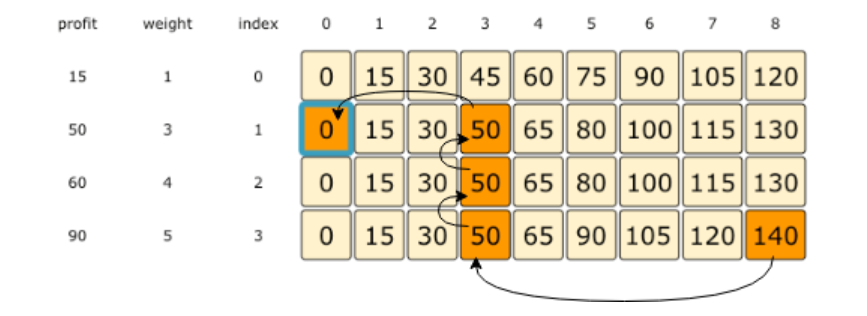
#### Find the selected items [**#**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/qV6RXWME4D3#find-the-selected-items)

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}.



1. [↑](#footnote-ref-1)