**Bottom-up Dynamic Programming**[#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/RM1BDv71V60#bottom-up-dynamic-programming)

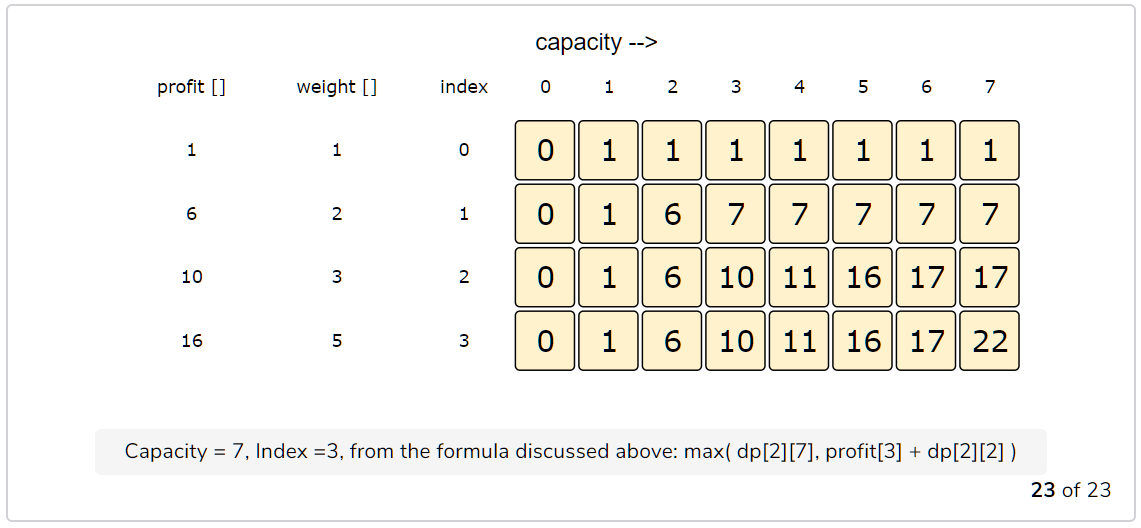
Let’s try to populate our dp[][] array from the above solution, working in a bottom-up fashion. Essentially, we want to find the maximum profit for every sub-array and for every possible capacity. **This means, dp[i][c] will represent the maximum knapsack profit for capacity ‘c’ calculated from the first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

So, for each item at index ‘i’ (0 <= i < items.length) and capacity ‘c’ (0 <= c <= 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], profit[i] + dp[i-1][c-weight[i]])



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

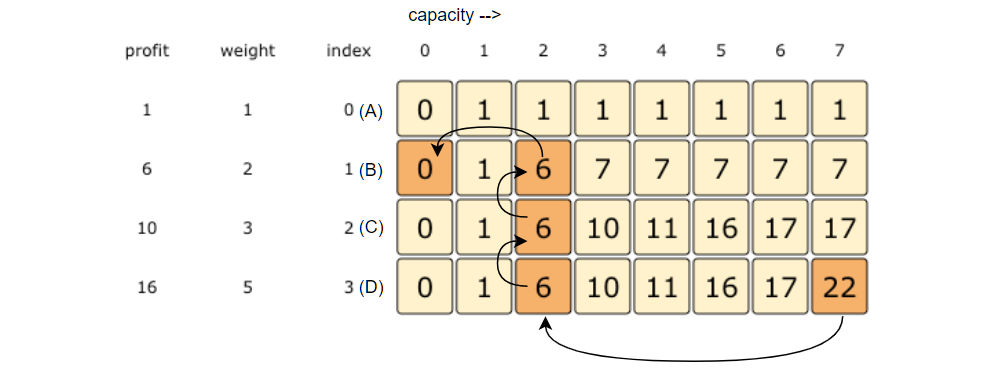
#### How to find the selected items? [**#**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/RM1BDv71V60#how-to-find-the-selected-items)

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:

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}.



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.

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, with '0' capacity we have '0' profit

    for(int i=0; i < n; i++)

      dp[i][0] = 0;

    // if we have only one weight, we will take it if it is not more than the capacity

    for(int c=0; c <= capacity; c++) {

      if(weights[0] <= c)

        dp[0][c] = profits[0];

    }

    // process all sub-arrays for all the capacities

    for(int i=1; i < n; i++) {

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

        int profit1= 0, profit2 = 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] = Math.max(profit1, profit2);

      }

    }

    printSelectedElements(dp, weights, profits, capacity);

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

    return dp[n-1][capacity];

  }

 private void printSelectedElements(int dp[][], int[] weights, int[] profits, int capacity){

   System.out.print("Selected weights:");

   int totalProfit = dp[weights.length-1][capacity];

   for(int i=weights.length-1; i > 0; i--) {

     if(totalProfit != dp[i-1][capacity]) {

       System.out.print(" " + weights[i]);

       capacity -= weights[i];

       totalProfit -= profits[i];

     }

   }

   if(totalProfit != 0)

     System.out.print(" " + weights[0]);

   System.out.println("");

 }

**2D Space optimized:**

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

    // basic checks

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

      return 0;

    int n = profits.length;

    // we only need one previous row to find the optimal solution, overall we need '2' rows

    // the above solution is similar to the previous solution, the only difference is that

    // we use `i%2` instead if `i` and `(i-1)%2` instead if `i-1`

    int[][] dp = new int[2][capacity+1];

    // if we have only one weight, we will take it if it is not more than the capacity

    for(int c=0; c <= capacity; c++) {

      if(weights[0] <= c)

        dp[0][c] = dp[1][c] = profits[0];

    }

    // process all sub-arrays for all the capacities

    for(int i=1; i < n; i++) {

      for(int c=0; c <= capacity; c++) {

        int profit1= 0, profit2 = 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] = Math.max(profit1, profit2);

      }

    }

    return dp[(n-1)%2][capacity];

  }

The above solution is similar to the previous solution, the only difference is that we use i%2 instead if i and (i-1)%2 instead if i-1. This solution has a space complexity of O(2\*C) = O(C)*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 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-weight[i]]

Since our inner loop is iterating over c:0-->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-weight[i]] might be overridden if “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:capacity-->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?

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

    // basic checks

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

      return 0;

    int n = profits.length;

    int[] dp = new int[capacity + 1];

    // if we have only one weight, we will take it if it is not more than the

    // capacity

    for (int c = 0; c <= capacity; c++) {

      if (weights[0] <= c)

        dp[c] = profits[0];

    }

    // process all sub-arrays for all the capacities

    for (int i = 1; i < n; i++) {

      for (int c = capacity; c >= 0; c--) {

        int profit1 = 0, profit2 = 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] = Math.max(profit1, profit2);

      }

    }

    return dp[capacity];

  }