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. Each item can only be selected once, as we don't have multiple quantities of any item.

Let's take an 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, Banana, Melon }
Weights: { 2, 3, 1, 4 }
Profits: { 4, 5, 3, 7 }
Knapsack Capacity: 5

Let's try to put different combinations of fruit in the knapsack, such that their total weights 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

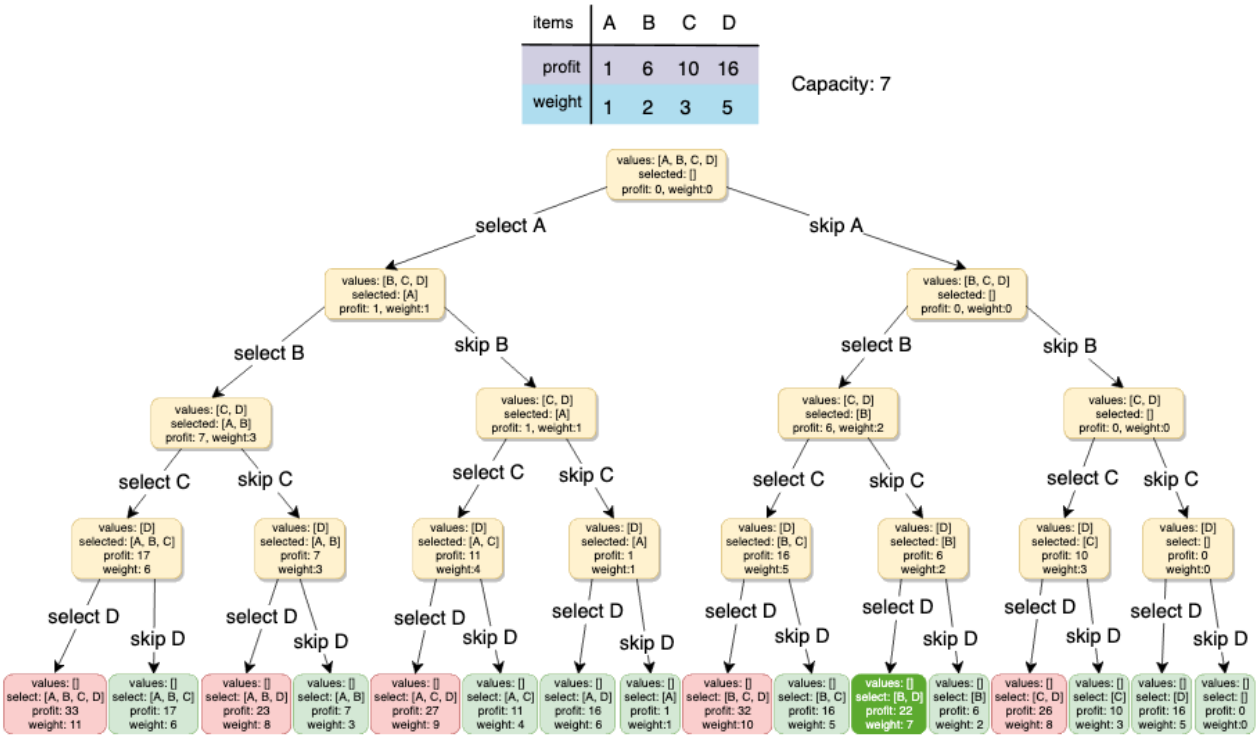
Given two integer arrays to represent weights and profits of 'N' items, we need to find a subset of these items which will give us the maximum profit such that their cumulative weight is not more than a given number 'C'. Each item can only be selected once, which means either we put an item in the knapsack or we skip it.

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, D), as shown in the diagram below. To try all combinations, our algorithm looks like:

```
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

    create a new set WITHOUT item 'i', and recursively process the
remaining items

return the set from the above two sets with higher profit
```

Here is a visual representation of our algorithm:



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 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.

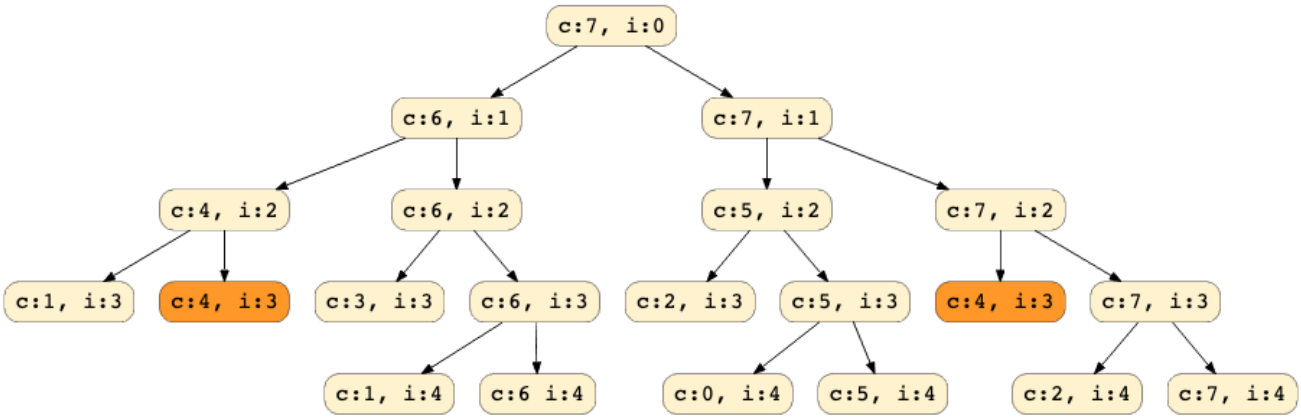
Code

```
let solveKnapsackBruteForce = (profits, weights, capacity) => {
    let knapSackRecursive = (capacity, currentIndex) => {
        // base checks
        if (capacity <= 0 || currentIndex >= profits.length) return 0;

        // recursive call after choosing the element at the current index
        // if the weight of the element at the current index exceeds
capacity, we shouldn't process this
        let profit1 = 0;
        if (weights[currentIndex] <= capacity) {
            profit1 = profits[currentIndex] + knapSackRecursive(capacity -
weights[currentIndex], currentIndex + 1);
        }

        // recursive call after excluding the element at the currentIndex
        const profit2 = knapSackRecursive(capacity, currentIndex + 1);

        return Math.max(profit1, profit2);
    };

    return knapSackRecursive(capacity, 0);
};

var profits = [1, 6, 10, 16];
var weights = [1, 2, 3, 5];
console.log(`Total knapsack profit: --->
${solveKnapsackBruteForce(profits, weights, 7)}`);
console.log(`Total knapsack profit: --->
${solveKnapsackBruteForce(profits, weights, 6)}`);
```

The time complexity of the above algorithm is 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.

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 weight 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 and we have overlapping subproblems even in this simple example. This 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:

```
let solveKnapsackTopDown = (profits, weights, capacity) => {
    const dp = [];
    let knapSackRecursive = (capacity, currentIndex) => {
        // base checks
        if (capacity <= 0 || currentIndex >= profits.length) return 0;

        dp[currentIndex] = dp[currentIndex] || [];
        if (typeof dp[currentIndex][capacity] !== 'undefined') {
            return dp[currentIndex][capacity];
        }

        // recursive call after choosing the element at the current index
        // if the weight of the element at the current index exceeds
capacity, we shouldn't process this
        let profit1 = 0;
        if (weights[currentIndex] <= capacity) {
            profit1 = profits[currentIndex] + knapSackRecursive(capacity -
weights[currentIndex], currentIndex + 1);
        }

        // recursive call after excluding the element at the currentIndex
        const profit2 = knapSackRecursive(capacity, currentIndex + 1);

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

    return knapSackRecursive(capacity, 0);
};

var profits = [1, 6, 10, 16];
var weights = [1, 2, 3, 5];
console.log(`Total knapsack profit: ---> ${solveKnapsackTopDown(profits,
weights, 7)}`);
console.log(`Total knapsack profit: ---> ${solveKnapsackTopDown(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 of all 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 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 `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 first 'i' items.**

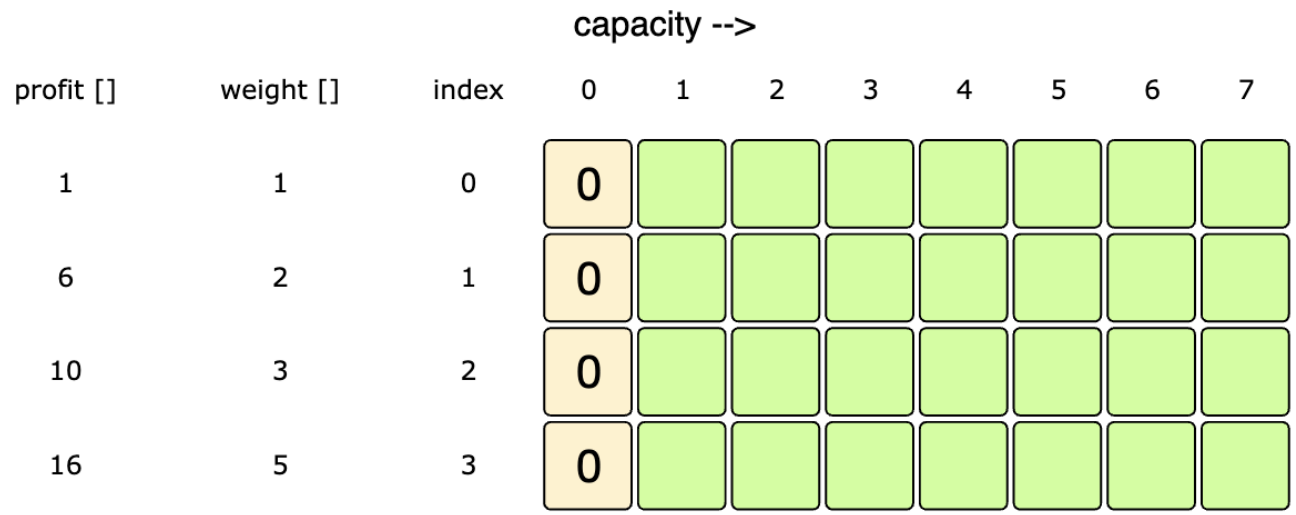
So, for each item at index 'i' (0 <= i < items.length) and capacity c (0 <= c < capacity), we have two options:

- Exclude the item at index 'i'. In this case, we will tak whatever profit we get from the subarray excluding this item => `dp[i-1][c]`.
- Include tthe item at index 'i' if it's weight is not more than the capacity. In this case, we include it's 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 the maximum of above two values.

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

Let's visually draw this and start with our base case of zero capacity:



If we consider the subarray till index 0, we only have one item to put in. We will take it if it is not more than the capacity.

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							
10	3	2	0							
16	5	3	0							

At capacity=1 and index=1, since the item at index=1 has weight 2, which is greater than the capacity, we will take `dp[index-1][capacity]`

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						
10	3	2	0							
16	5	3	0							

Next, we fill up the table using the formula above: `max(dp[0][2], profits[1] + dp[0][0])`

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					
10	3	2	0							
16	5	3	0							

The complete table:

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

```
let solveKnapsack = function (profits, weights, capacity) {
    const n = profits.length;
    if (capacity <= 0 || n === 0 || weights.length !== n) return 0;

    const dp = Array(n)
        .fill(0)
        .map(() => Array(capacity + 1).fill(0));

    // populate the capacity=0 column with 0 profit
    for (let 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 (let c = 0; c <= capacity; c++) {
        if (weights[0] <= c) dp[0][c] = profits[0];
    }

    //process all subarrays for all capacities
    for (let i = 1; i < n; i++) {
        for (let c = 1; c <= capacity; c++) {
            let profit1 = 0;
            let profit2 = 0;

            // include the item if its not more than the capacity
            if (weights[i] <= c) profit1 = profits[i] + dp[i - 1][c -
weights[i]];

            //exclude the ittem
            profit2 = dp[i - 1][c];

            //take max
            dp[i][c] = Math.max(profit1, profit2);
        }
    }

    return dp[n - 1][capacity];
};

var profits = [1, 6, 10, 16];
var weights = [1, 2, 3, 5];
console.log(`Total knapsack profit: ---> ${solveKnapsack(profits, weights,
7)}`);
console.log(`Total knapsack profit: ---> ${solveKnapsack(profits, weights,
6)}`);
```

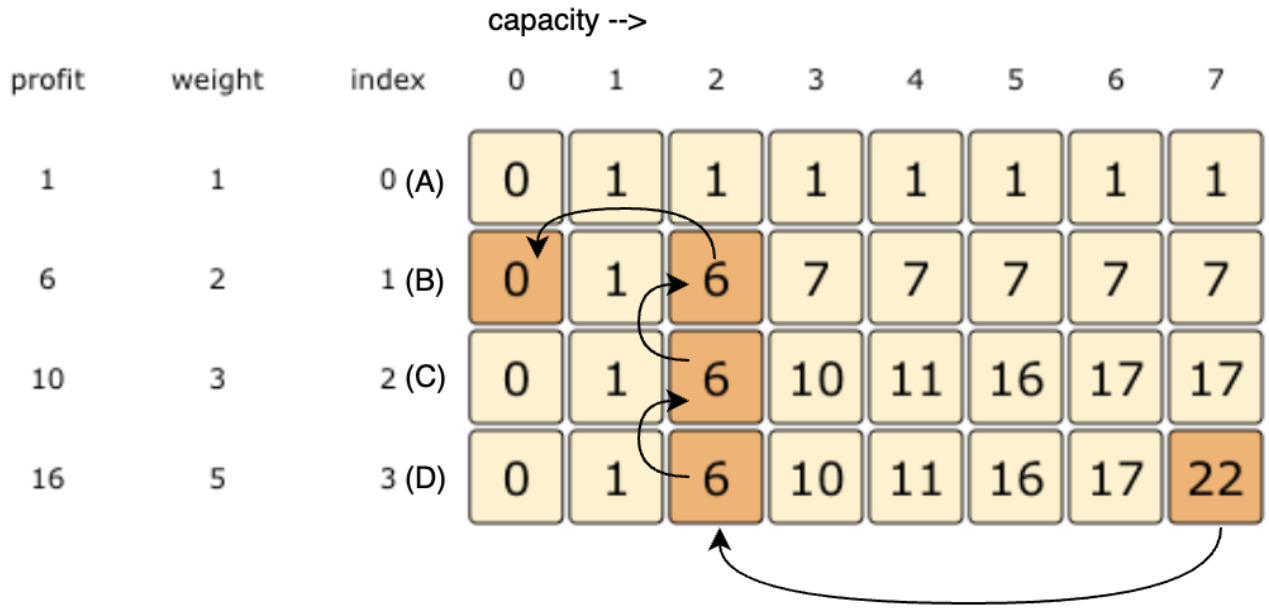
The above solution has time and space complexity of `O(N*C)`, where 'N' represents total items and 'C' is the maximum capacity.

How do we 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}.

Let's write a function to print the set of items going in the knapsack:

```
let solveKnapsack = function (profits, weights, capacity) {
    const n = profits.length;
    if (capacity <= 0 || n === 0 || weights.length !== n) return 0;

    const dp = Array(n)
        .fill(0)
        .map(() => Array(capacity + 1).fill(0));

    // populate the capacity=0 column with 0 profit
    for (let 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 (let c = 0; c <= capacity; c++) {
        if (weights[0] <= c) dp[0][c] = profits[0];
    }

    //process all subarrays for all capacities
    for (let i = 1; i < n; i++) {
        for (let c = 1; c <= capacity; c++) {
            let profit1 = 0;
            let profit2 = 0;

            // include the item if its not more than the capacity
            if (weights[i] <= c) profit1 = profits[i] + dp[i - 1][c -
weights[i]];

            //exclude the ittem
            profit2 = dp[i - 1][c];

            //take max
            dp[i][c] = Math.max(profit1, profit2);
        }
    }

    // This block is to just print the selected knapsack items
    let selectedWeights = '';
    let totalProfit = dp[n - 1][capacity];
    let remainingCapacity = capacity;
    for (let i = n - 1; i > 0; i++) {
        if (totalProfit !== dp[i - 1][remainingCapacity]) {
            selectedWeights = `${weights[i]} ${selectedWeights}`;
            remainingCapacity -= weights[i];
            totalProfit -= profits[i];
        }
    }

    if (totalProfit !== 0) selectedWeights = `${weights[0]}
${selectedWeights}`;
```

```
    console.log(`Selected weights: ${selectedWeights}`);

    return dp[n - 1][capacity];
};

var profits = [1, 6, 10, 16];
var weights = [1, 2, 3, 5];
console.log(`Total knapsack profit: ---> ${solveKnapsack(profits, weights,
7)}`);
console.log(`Total knapsack profit: ---> ${solveKnapsack(profits, weights,
6)}`);
```

Optimizing Further

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

```
let solveKnapsack = function(profits, weights, capacity) {
  const n = profits.length;
  if (capacity <= 0 || n == 0 || weights.length != n) return 0;

  // 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`
  const dp = Array(2)
    .fill(0)
    .map(() => Array(capacity + 1).fill(0));

  // if we have only one weight, we will take it if it is not more than
the capacity
  for (let 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 (let i = 1; i < n; i++) {
    for (let c = 1; c <= capacity; c++) {
      let 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);
    }
  }

  // maximum profit will be at the bottom-right corner.
  return dp[(n - 1) % 2][capacity];
};

var profits = [1, 6, 10, 16];
var weights = [1, 2, 3, 5];
console.log(`Total knapsack profit: ---> ${solveKnapsack(profits, weights,
7)}`);
console.log(`Total knapsack profit: ---> ${solveKnapsack(profits, weights,
6)}`);
```

The 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 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.

```
let solveKnapsack = function (profits, weights, capacity) {
    const n = profits.length;
    if (capacity <= 0 || n == 0 || weights.length != n) return 0;

    const dp = Array(capacity + 1).fill(0);

    for (let c = 0; c <= capacity; c++) {
        if (weights[0] <= c) dp[c] = profits[0];
    }

    // process all sub-arrays for all the capacities
    for (let i = 1; i < n; i++) {
        for (let c = capacity; c >= 0; c--) {
            let 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];
};

var profits = [1, 6, 10, 16];
var weights = [1, 2, 3, 5];
console.log(`Total knapsack profit: ---> ${solveKnapsack(profits, weights,
7)}`);
console.log(`Total knapsack profit: ---> ${solveKnapsack(profits, weights,
6)}`);
```