0/1 Knapsack (medium)

We'll cover the following

Introduction

Given the weights and profits of 'N' items, we are asked to put these items in a knapsack with a capacity 'C.' The goal is to get the maximum profit out of the knapsack items. Each item can only be selected once, as we don't have multiple quantities of any item.

Let's take Merry's example, 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 various combinations of fruits in the knapsack, such that their total weight 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 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.

Try it yourself

Try solving this question here:

Java | Python3 | C++ | JS

```
class Knapsack {

  public int solveKnapsack(int[] profits, int[] weights, int capacity) {
    // TODO: Write your code here
    return -1;
  }

  public static void main(String[] args) {
    Knapsack ks = new Knapsack();
    int[] profits = {1, 6, 10, 16};
    int[] weights = {1, 2, 3, 5};
    int maxProfit = ks.solveKnapsack(profits, weights, 7);
```

```
    System.out.println("Total knapsack profit ---> " + maxProfit);
    maxProfit = ks.solveKnapsack(profits, weights, 6);
    System.out.println("Total knapsack profit ---> " + maxProfit);
  }
}
```

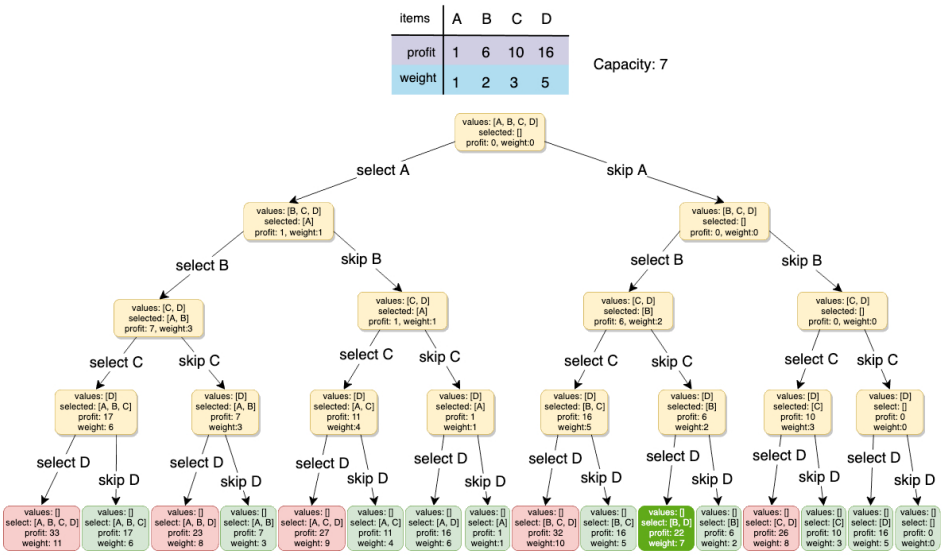
Run Save Reset

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, and D), as shown in the diagram below. To try all the combinations, our algorithm will look 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 that is 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

Here is the code for the brute-force solution:

Java Python3 C++ JS

```
class Knapsack {

  public int solveKnapsack(int[] profits, int[] weights, int capacity) {
    return this.knapsackRecursive(profits, weights, capacity, 0);
  }

  private int knapsackRecursive(int[] profits, int[] weights, int capacity, int currentIndex) {
    // base checks
    if (capacity <= 0 || currentIndex >= profits.length)
      return 0;

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

    // recursive call after excluding the element at the currentIndex
    int profit2 = knapsackRecursive(profits, weights, capacity, currentIndex + 1);

    return Math.max(profit1, profit2);
  }

```

```
public static void main(String[] args) {
  Knapsack ks = new Knapsack();
  int[] profits = {1, 6, 10, 16};
  int[] weights = {1, 2, 3, 5};
```

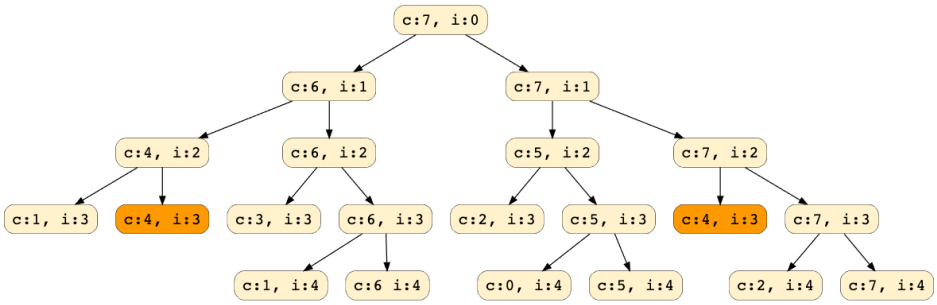
Run Save Reset

Time and Space complexity

The above algorithm's time complexity is exponential $O(2^n)$, where 'n' represents the total number of items. This can also be confirmed from the above recursion tree. As we can see, 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 the recursive algorithm works in a depth-first fashion, which means that we can't have more than 'n' recursive calls on the call stack at any time.

Overlapping Sub-problems: 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 weights 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. Hence we have an overlapping sub-problems pattern. We can use Memoization to solve overlapping sub-problems efficiently.

Top-down Dynamic Programming with Memoization

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 every possible capacity 'c.'

Code

Here is the code with memoization (see changes in the highlighted lines):

Java | Python3 | C++ | JS

```
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 || currentIndex >= profits.length)
      return 0;

    // if we have already solved a similar problem, return the result from memory
    if(dp[currentIndex][capacity] != null)
      return dp[currentIndex][capacity];

    // recursive call after choosing the element at the currentIndex
    // if the weight of the element at currentIndex exceeds the capacity, we shouldn't process this
    int profit1 = 0;
    if( weights[currentIndex] <= capacity )
      profit1 = profits[currentIndex] + knapsackRecursive(dp, profits, weights,
          capacity - weights[currentIndex], currentIndex + 1);

    // recursive call after excluding the element at the currentIndex
    int profit2 = knapsackRecursive(dp, profits, weights, capacity, currentIndex + 1);

```

Run Save Reset

Time and Space complexity

Since our memoization array `dp[profits.length][capacity+1]` stores the results for 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 use $O(N * C)$ 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 by working in a bottom-up fashion. Essentially, we want to find the maximum profit for every sub-array and every possible capacity. **This means that `dp[i][c]` will represent the maximum knapsack profit for capacity 'c' calculated from the first 'i' items.**

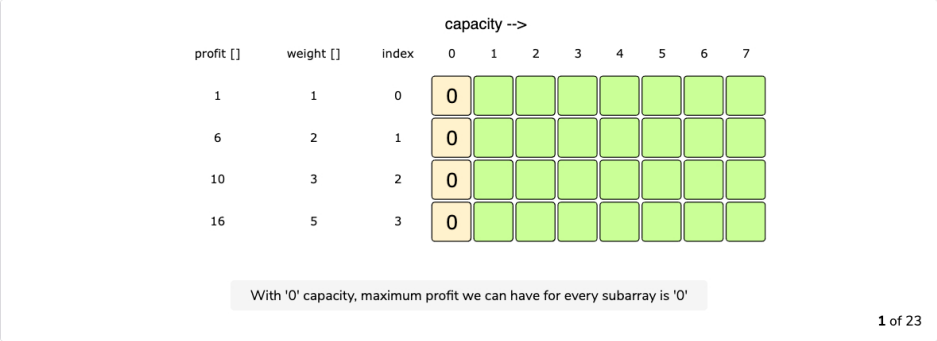
So, for each item at index 'i' (0 <= i < items.length) and capacity 'c' (0 <= c <= capacity), we have two options:

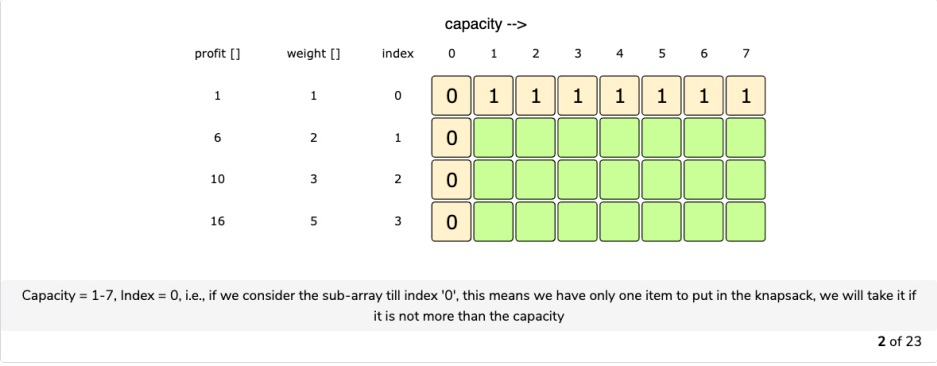
1. Exclude the item at index 'i.' In this case, we will take whatever profit we get from the sub-array excluding this item => `dp[i-1][c]`
2. Include the item at index 'i' if its weight is not more than the capacity. In this case, we include its profit plus whatever profit we get from the remaining capacity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

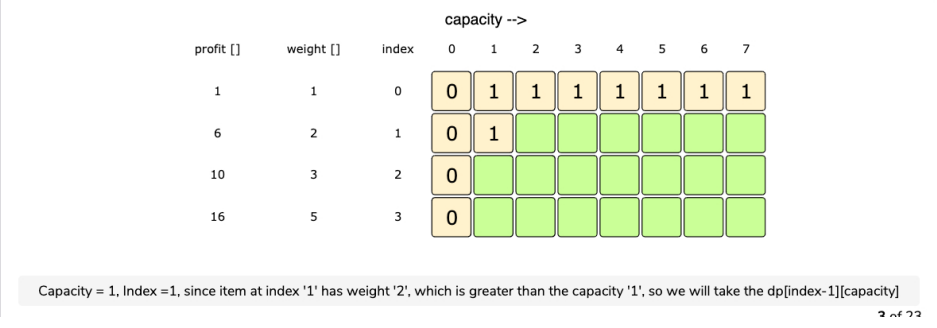
Finally, our optimal solution will be maximum of the above two values:

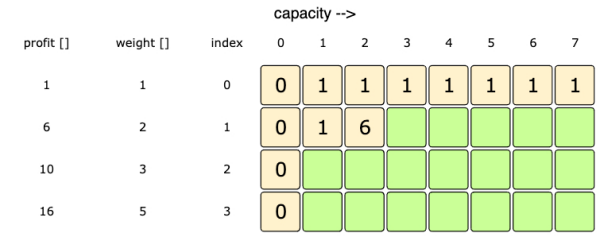
```
dp[i][c] = max (dp[i-1][c], profit[i] + dp[i-1][c-weight[i]])
```

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

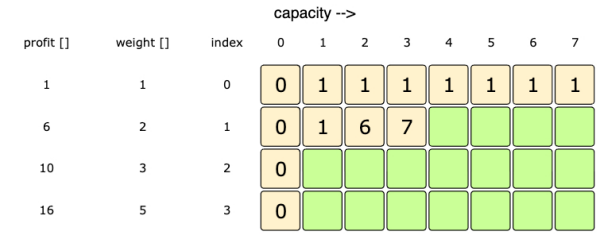




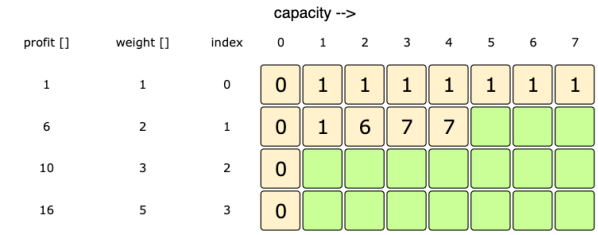




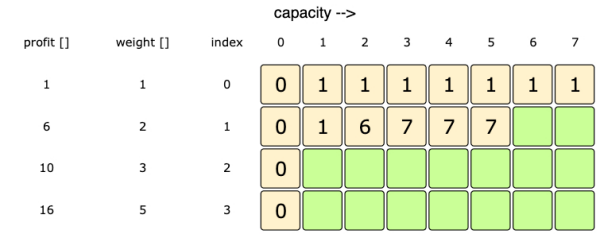
Capacity = 2, Index =1, from the formula discussed above: max(dp[0][2], profit[1] + dp[0][0])



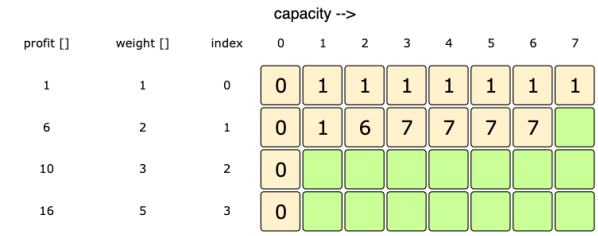
Capacity = 3, Index =1, from the formula discussed above: max(dp[0][3], profit[1] + dp[0][1])



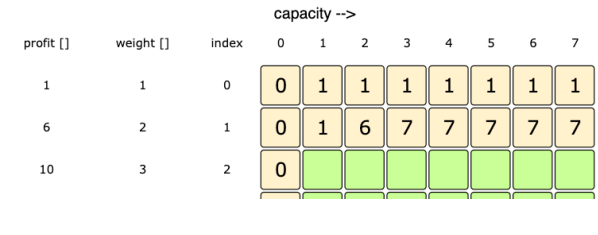
Capacity = 4, Index =1, from the formula discussed above: max(dp[0][4], profit[1] + dp[0][2])



Capacity = 5, Index =1, from the formula discussed above: max(dp[0][5], profit[1] + dp[0][3])



Capacity = 6, Index =1, from the formula discussed above: max(dp[0][6], profit[1] + dp[0][4])



16	5	3	0						

Capacity = 7, Index =1, from the formula discussed above: max(dp[0][7], profit[1] + dp[0][5])

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						
16	5	3	0							

Capacity = 1, Index =2, since item at index '2' has weight '3', which is greater than the capacity '1', so we will take the 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	6	7	7	7	7	7
10	3	2	0	1	6					
16	5	3	0							

Capacity = 2, Index =2, since item at index '2' has weight '3', which is greater than the capacity '1', so we will take the 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	6	7	7	7	7	7
10	3	2	0	1	6	10				
16	5	3	0							

Capacity = 3, Index =2, from the formula discussed above: max(dp[1][3], profit[2] + dp[1][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	7	7	7	7	7
10	3	2	0	1	6	10	11			
16	5	3	0							

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

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		
16	5	3	0							

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

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
10	3	2	0	1	6	10	11	16	17
16	5	3	0						

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

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							

Capacity = 7, Index =2, from the formula discussed above: max(dp[1][7], profit[2] + dp[1][4])

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						

Capacity = 1, Index =3, since item at index '3' has weight '5', which is greater than the capacity '1', so we will take the 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	6	7	7	7	7	7
10	3	2	0	1	6	10	11	16	17	17
16	5	3	0	1	6					

Capacity = 2, Index =3, since item at index '3' has weight '5', which is greater than the capacity '2', so we will take the dp[index-1][capacity]

capacity -->

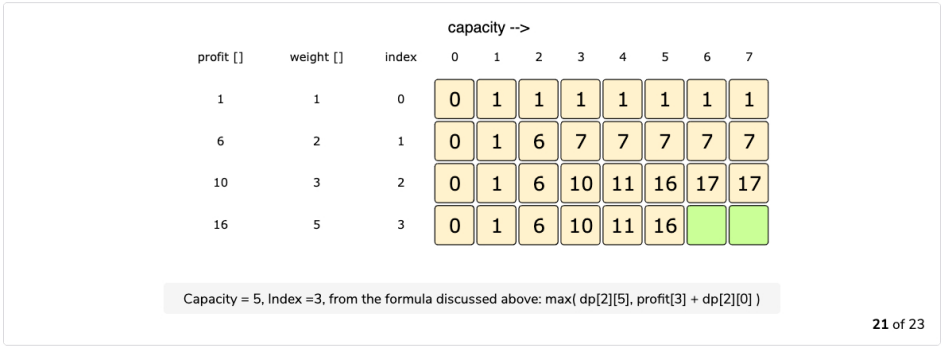
profit []	weight []	index	0	1	2	3	4	5	6	7
1	1	0	0	1	1	1	1	1	1	1
6	2	1	0	1	6	7	7	7	7	7
10	3	2	0	1	6	10	11	16	17	17
16	5	3	0	1	6	10				

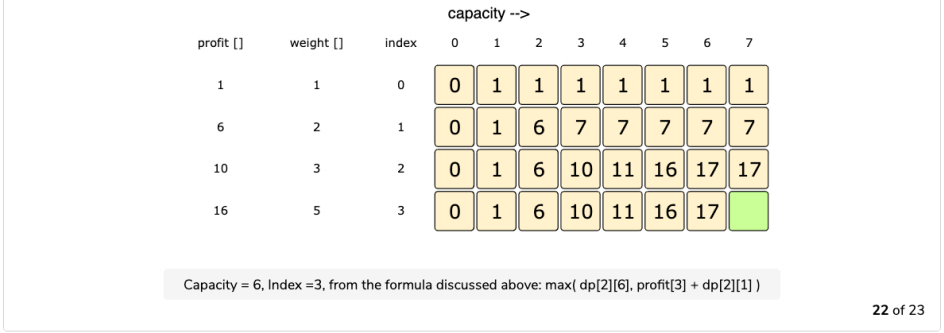
Capacity = 3, Index =3, since item at index '3' has weight '5', which is greater than the capacity '3', so we will take the dp[index-1][capacity]

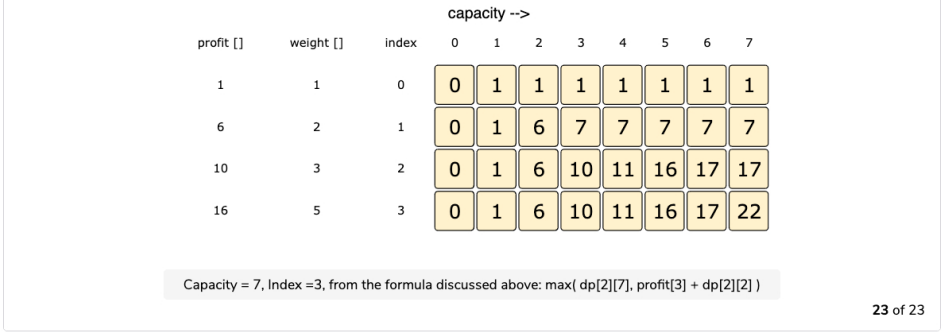
capacity -->

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	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			

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







Code

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

Java Python3 C++ JS

```java
class Knapsack {

  public 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[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
```

Run Save Reset

Time and Space complexity

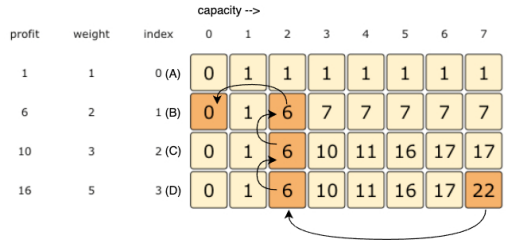
The above solution has the time and space complexity of $O(N * C)$, where 'N' represents total items, and 'C' is the maximum capacity.

How can we find the selected items?

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

As you remember, 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:



1. '22' did not come from the top cell (which is 17); hence we must include the item at index '3' (which is 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 from the top cell, so we must include this item (which is item 'B').
6. 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.
7. Thus, the items going into the knapsack are {B, D}.

Let's write a function to print the set of items included in the knapsack.

Java | Python3 | C++ | JS

```java
import java.util.*;

class 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)
```

Run | Save | Reset

Challenge

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

Hide Hint

We only need one previous row to find the optimal solution!

Java | Python3 | C++ | JS

```java
class Knapsack {

  static int solveKnapsack(int[] profits, int[] weights, int capacity) {
    //TODO: Write - Your - Code
    return -1;
  }
}
```

Test | Hide Solution | Save | Reset

Solution

```java
class Knapsack {

  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];
```

The solution above 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 knapsack's maximum capacity.

This space optimization solution can also be implemented using a single array. It is a bit tricky, 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 again in the current iteration.

Can you try writing this algorithm?

Java | Python3 | C++ | JS

```java
class Knapsack {

  static int solveKnapsack(int[] profits, int[] weights, int capacity) {
    //TODO: Write - Your - Code
    return -1;
  }
}
```

Test | Hide Solution | Save | Reset

Solution

```java
class Knapsack {

  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);
```


Back

Introduction

Next

Equal Subset Sum Partition (medium)

Mark as Completed