0/1 Knapsack (medium)

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 out of 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 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, 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

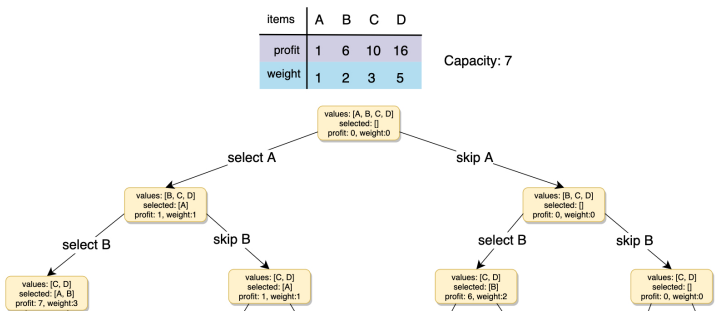
Given two integer arrays to represent weights and profits of 'N' items, we need to find a subset of these items which will give us maximum profit such that their cumulative weight is not more than a given number 'C'. Each item can only be selected once, which means either we put an item in the knapsack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

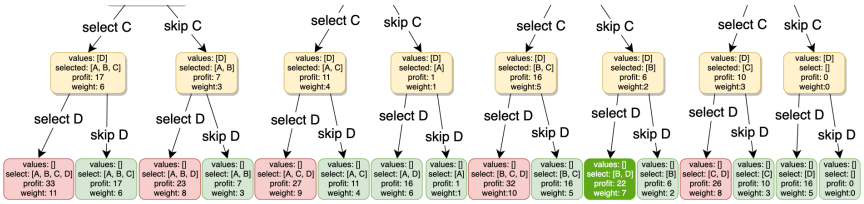
Basic Solution

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

```
let solveKnapsack = function(profits, weights, capacity) {
  function knapsackRecursive(profits, weights, capacity, 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
    let 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
    const profit2 = knapsackRecursive(profits, weights, capacity, currentIndex + 1);

    return Math.max(profit1, profit2);
  }

  return knapsackRecursive(profits, weights, capacity, 0);
};

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)}`);
```

RUN | SAVE | RESET

Close

Output 2.006s

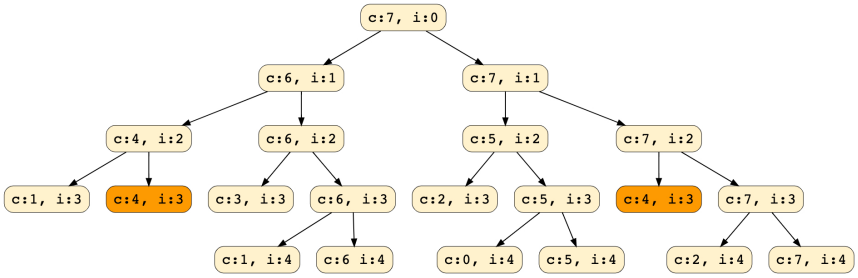
```
Total knapsack profit: ---> 22
Total knapsack profit: ---> 17
```

Time and Space complexity

The time complexity of the above algorithm 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 efficiently solve overlapping sub-problems.

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

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

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

Java | Python3 | C++ | JS

```
let solveKnapsack = function(profits, weights, capacity) {
  const dp = [];

  function knapsackRecursive(profits, weights, 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 currentIndex
    // if the weight of the element at currentIndex exceeds the capacity, we shouldn't process this
    let 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
    const profit2 = knapsackRecursive(profits, weights, capacity, currentIndex + 1);

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

```

RUN | SAVE | RESET

Close

Output 1.821s

```
Total knapsack profit: ---> 22
Total knapsack profit: ---> 17
```

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 for every possible capacity. **This means that `dp[i][c]` will represent the maximum knapsack profit for capacity 'c' calculated from the first 'i' items.**

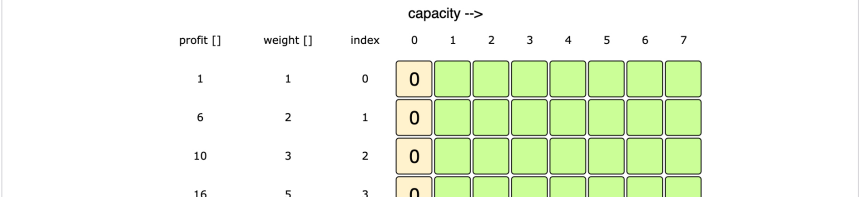
So, for each item at index 'i' (0 <= i < items.length) and capacity 'c' (0 <= c <= capacity), we have two options:

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:



With '0' capacity, maximum profit we can have for every subarray is '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							
10	3	2	0							
16	5	3	0							

Capacity = 1-7, Index = 0, i.e., if we consider the sub-array till index '0', this means we have only one item to put in the knapsack, 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	1						
10	3	2	0							
16	5	3	0							

Capacity = 1, Index =1, since item at index '1' has weight '2', 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					
10	3	2	0							
16	5	3	0							

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

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

Capacity = 4, Index =1, from the formula discussed above: max(dp[0][4], profit[1] + dp[0][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		
10	3	2	0							
16	5	3	0							

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

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

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		

Capacity = 5, Index =3, from the formula discussed above: max(dp[2][5], profit[3] + dp[2][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	17	17
16	5	3	0	1	6	10	11	16	17	

Capacity = 6, Index =3, from the formula discussed above: max(dp[2][6], profit[3] + dp[2][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	17	17
16	5	3	0	1	6	10	11	16	17	22

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

Code

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

Java | Python3 | C++ | JS

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

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

  // populate the capacity=0 columns; with '0' capacity we have '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 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][c - weights[i]];
      // exclude the item
      profit2 = dp[i - 1][c];
      // take maximum
      dp[i][c] = Math.max(profit1, profit2);
    }
```

RUN SAVE RESET

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

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

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

  // populate the capacity=0 columns; with '0' capacity we have '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 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][c - weights[i]];
      // exclude the item
      profit2 = dp[i - 1][c];
      // take maximum
      dp[i][c] = Math.max(profit1, profit2);
    }
```

RUN SAVE RESET

Close

Output 2.234s

```
Selected weights: 2 5
Total knapsack profit: ---> 22
Selected weights: 1 2 3
Total knapsack profit: ---> 17
```

Challenge

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

Show Hint

Java Python3 C++ JS

```
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);
    }
```

RUN SAVE RESET

Close

Output 3.526s

```
Total knapsack profit: ---> 22
Total knapsack profit: ---> 17
```

The solution above is similar to the previous solution, the only difference is that we use `i%2` instead if `i` and

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 maximum capacity of the knapsack.

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

```
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);

  // 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[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);
    }
  }

  // maximum profit will be at the bottom-right corner.
  return dp[capacity];
};
```

RUN | SAVE | RESET

Close

Output 1.739s

```
Total knapsack profit: ---> 22
Total knapsack profit: ---> 17
```

Interviewing soon? We've partnered with Hired so that companies apply to you instead of you applying to them. See how

Explore
Tracks
My Courses
Edpresso
Refer a Friend
Create