



Knapsack Problem



The problem

Input

- We have a set of items; each item have a weight and a value
- We also have a container that can carry up to a specific total weight (maximum capacity).

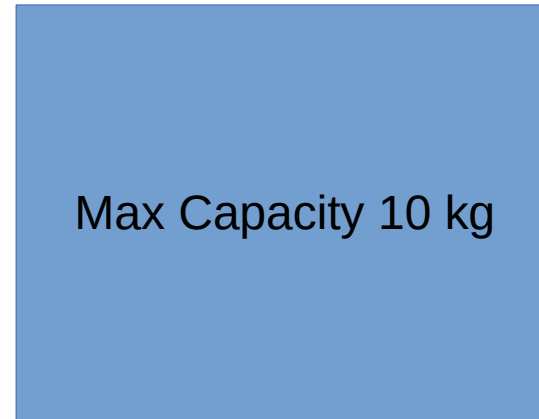
Objective

- The objective is to find a subset of the items that can fit in the container (meaning their total weight will be less or equal to the capacity of the container) while maximizing the total value.
- Each item can either be contained or not (meaning we cannot use fractional quantities).

Assume we have the following items

Index	Weight	Value
1	8 kg	\$17
2	4 kg	\$12
3	3 kg	\$6

And this bag



All possible items combinations

Items	Weight	Value
1	8 kg	\$17
2	4 kg	\$12
3	3 kg	\$6
1 - 2	12 kg	\$29
1 - 3	11 kg	\$23
2 - 3	7 kg	\$18
1 - 2 - 3	19 kg	\$35

Maximum value while weight less than 10 is obtained by combining items 2 and 3

2 - 3	7 kg	\$18
-------	------	------

This solution creates all possible item combinations have exponential complexity $O(2^n)$

A better algorithm can be created using Dynamic programming.

Solving Knapsack using dynamic programming

Assume we have the following items sorted by their weight

The knapsack has a maximum capacity of 10 kg

Index	Weight	Value
0	0	0
1	1	2
2	3	6
3	4	12
4	5	12
5	8	17

Max Capacity 10 kg



Solving Knapsack using dynamic programming

TO KEEP THINGS SIMPLE

- * WE HAVE ADDED AN ITEM WITH WEIGHT AND VALUE EQUAL TO 0
- * WE KEEP THE ITEMS SORTED BY VALUE IN DECREASING ORDER

Solving Knapsack using dynamic programming

* Including the 0 weight item we have added we have $N = 6$ items altogether while we have 11 possible discrete weights until we reach the full capacity of 10.

* For each item (and including the 0 capacity) we create $11 \times 6 = 66$ new bags for each possible capacity and initialize it to 0 as can be seen here:

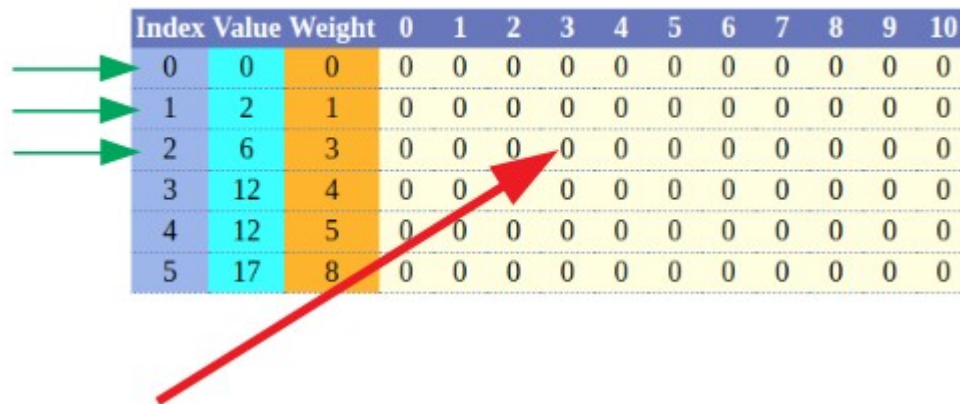
Index	Value	Weight	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	0	0	0	0	0	0	0	0	0	0
2	6	3	0	0	0	0	0	0	0	0	0	0	0
3	12	4	0	0	0	0	0	0	0	0	0	0	0
4	12	5	0	0	0	0	0	0	0	0	0	0	0
5	17	8	0	0	0	0	0	0	0	0	0	0	0

Think of each element of the array as a bag whose capacity equal to

Solving Knapsack using dynamic programming

Think of each element of the array as a bag whose capacity equal to its column index.

Each row in the array has access ONLY to the item for it and to those that occur before it as can be seen in the following example:



Index	Value	Weight	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	0	0	0	0	0	0	0	0	0	0
2	6	3	0	0	0	0	0	0	0	0	0	0	0
3	12	4	0	0	0	0	0	0	0	0	0	0	0
4	12	5	0	0	0	0	0	0	0	0	0	0	0
5	17	8	0	0	0	0	0	0	0	0	0	0	0

The element where the red arrow points is a bag with capacity 3 while it has access to items 0 – 1 - 2

Solving Knapsack using dynamic programming

Our objective is to fill the 6 X 11 array with the maximum possible value that each of the “individual” bag can hold.

We can only use items from the same or previous rows but not from the next.

Once the value for a bag is set it will remain the same and we will be able to access it any time.

If we fill all the elements of the array the last one will hold the maximum cumulative value that does not exceed the maximum capacity. For example in the following table the largest value will be the value of the green element (which can be also thought as a bag with maximum capacity 10).

Index	Value	Weight	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	0	0	0	0	0	0	0	0	0	0
2	6	3	0	0	0	0	0	0	0	0	0	0	0
3	12	4	0	0	0	0	0	0	0	0	0	0	0
4	12	5	0	0	0	0	0	0	0	0	0	0	0
5	17	8	0	0	0	0	0	0	0	0	0	0	0

Solving Knapsack using dynamic programming

Filling the first row is obvious.

Since both weight and value is 0 we just set all elements to it:

Index	Value	Weight	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0

Solving Knapsack using dynamic programming

To fill the second row lets start from the first bag which has a weight of 1

The first element of the second row will look as follows:

Index	Value	Weight	0	1
0	0	0	0	0
1	2	1	0	2

Note that when we reach the bag with capacity 1 and since it can fit item1 whose weight is 1 we have changed the element's value to 2 (which is the value of the item).

At this point we are sure that with only item1 the bag with capacity 1 has a maximum value of 2 while it contains this item; this will not be changed any more while the algorithm is running

Following the same logic we complete the full row which upon completion will look as follows:

Index	Value	Weight	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2	2	2	2	2	2	2	2	2

Solving Knapsack using dynamic programming

To fill the third row we follow exactly the same procedure as before

The first two elements of the third row will look as follows:

Index	Value	Weight	0	1	2
0	0	0	0	0	0
1	2	1	0	2	2
2	6	3	0	2	2

Since the item weight is 3 and it does not fit in the bag (whose capacity is 2) we keep it as it is so the values for bags With 1 and 2 capacity remain 2.

Things become more interesting as we move to the bag with capacity 3; the item now fits in the bag and if we remove its current contents (which have a value of 2) the value becomes 6 while there is no more remaining capacity so since 6 is larger than 2 we leave it in the bag and continue:

Index	Value	Weight	0	1	2	3
0	0	0	0	0	0	0
1	2	1	0	2	2	2
2	6	3	0	2	2	6

Solving Knapsack using dynamic programming

Now we need to fill the bag with capacity 4 of the third row. Currently our table looks as follows:

Index	Value	Weight	0	1	2	3	4
0	0	0	0	0	0	0	0
1	2	1	0	2	2	2	2
2	6	3	0	2	2	6	?

The question is what should the value of the bag with capacity 4 be assuming we can use the available items (1 and 2).

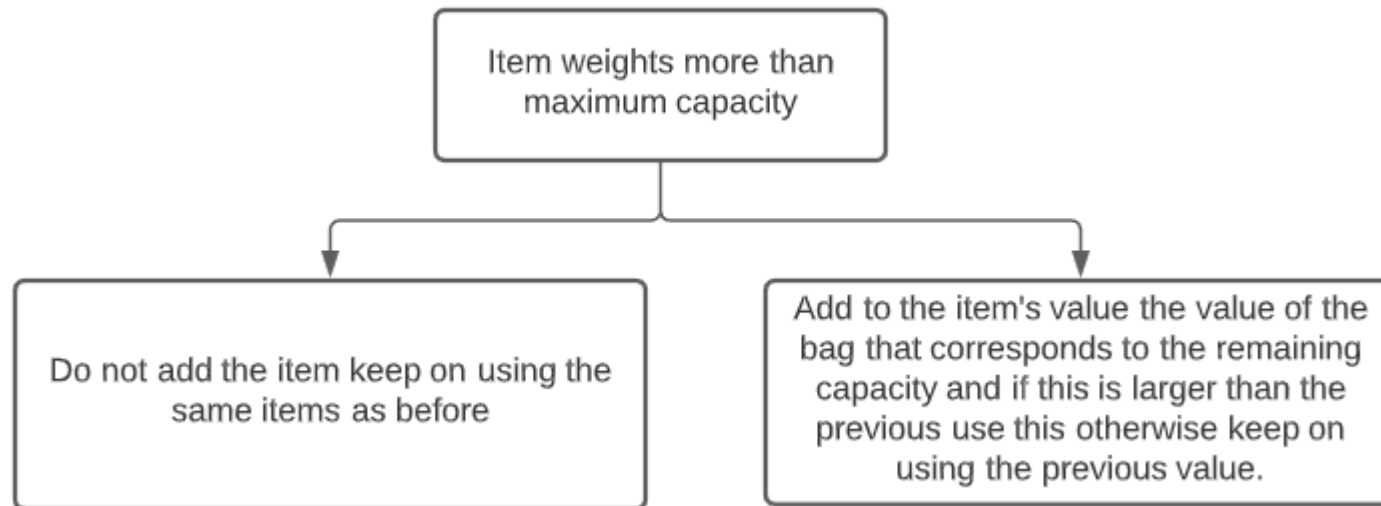
Item 2 weighs 3 kg meaning that it fits the bag. If we replace the contents of the bag with item 2 its value becomes 6 while it has a remaining capacity of 1 (since it can hold up to 4 kg and item 2 weighs 3 kg).

We already know that the maximum capacity for a bag with a capacity of 1kg that is using all the available items except item2 is 2; to find it we simply go to the element that exists in the previous row (1 in this case) and lookup the capacity for a bag with Capacity 1. So the maximum capacity for the bag 4 will become $6 + 2 = 8$:

Index	Value	Weight	0	1	2	3	4
0	0	0	0	0	0	0	0
1	2	1	0	2	2	2	2
2	6	3	0	2	2	6	8

Solving Knapsack using dynamic programming

At this point we know enough to create a generic algorithm to add a item to a specific element (a bag with known element):



Or is pseudo code:

Where:

i : The item index

j: The bag index

Bag: The bags array

```
If item[i].weight > j:
    Bag[i,j] = Bag[i-1, j]
else:
    remaining_capacity = j - item[i].weight
    v1 = item[i].v + Bag[i-1, remaining_capacity]
    Bag[i,j] = max(v1, Bag[i-1, j])
```

Solving Knapsack using dynamic programming

Continuing the same process as before and applying the algorithm we have expressed we end up with the following table for all the items:

Value	Weight	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	2	2	2	2	2	2	2	2	2	2
6	3	0	2	2	6	8	8	8	8	8	8	8
12	4	0	2	2	6	12	14	14	18	20	20	20
12	5	0	2	2	6	12	14	14	18	20	24	26
17	8	0	2	2	6	12	14	14	18	20	24	26

Meaning that using the given items and having a capacity of 10 the maximum value we can carry in the bag is 26.

Although we know have the maximum value we still need to find out the corresponding items.

Solving Knapsack using dynamic programming

Finding the items subset that maximizes the value

Index	Value	Weight	Selected	0	1	2	3	4	5	6	7	8	9	10
0	0	0		0	0	0	0	0	0	0	0	0	0	0
1	2	1	✓	0	2	2	2	2	2	2	2	2	2	2
2	6	3		0	2	2	6	8	8	8	8	8	8	8
3	12	4	✓	0	2	2	6	12	14	14	18	20	20	20
4	12	5	✓	0	2	2	6	12	14	14	18	20	24	26
5	17	8		0	2	2	6	12	14	14	18	20	24	26

Starting from the last item of the array we compare its value with the value of the previous element in the same bag.

If these values are the same this means that the item is not selected and continue with the previous.

Otherwise: Mark the item as selected, subtract the item's weight from the current bags maximum capacity and move to the element of this capacity in the previous row.

Follow the same procedure until the accumulated value of the selected items equals the maximum value that can be found in the last item of the array.