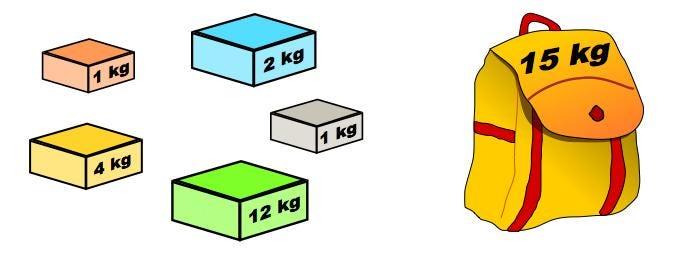
# **How to solve the Knapsack Problem with dynamic programming**



***Update:*** *Read about optimizing the space complexity of the dynamic programming solution in my follow-up article* [*here*](https://medium.com/@fabianterh/optimizing-the-knapsack-problem-dynamic-programming-solution-for-space-complexity-c6bcdff3870b)*.*

The [Knapsack Problem](https://en.wikipedia.org/wiki/Knapsack_problem) is a really interesting problem in combinatorics — to cite Wikipedia,

“given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.”

From Wikipedia, we see that there are a few variations of the Knapsack Problem: 0–1 knapsack, bounded knapsack, and unbounded knapsack. Today, we’ll be focusing on the most common (and simplest) variation: **the 0–1 knapsack problem.**

A slightly more interesting and relatable phrasing of the 0–1 knapsack problem is:

Consider a thief gets into a home to rob and he carries a knapsack. There are fixed number of items in the home — each with its own weight and value — Jewelry, with less weight and highest value vs tables, with less value but a lot heavy. To add fuel to the fire, the thief has an old knapsack which has limited capacity. Obviously, he can’t split the table into half or jewelry into 3/4ths. He either takes it or leaves it. [Source](https://www.hackerearth.com/practice/notes/the-knapsack-problem/)

Unfortunately, I had some difficulty understanding some parts of the Hackerearth article, which is why I decided to write my own article. This article will be largely based off Hackerearth’s article and code snippets are written in Java. I’ll be tacking on additional explanations and elaborations where I feel they are necessary.

# **Dynamic programming**

We’ll be solving this problem with dynamic programming. Dynamic programming requires an optimal substructure and overlapping sub-problems, both of which are present in the 0–1 knapsack problem, as we shall see.

It’s fine if you don’t understand what “optimal substructure” and “overlapping sub-problems” are (that’s an article for another day). Essentially, it just means a particular *flavor* of problems that allow us to reuse previous solutions to smaller problems in order to calculate a solution to the current problem. You’ll see what I mean in a bit.

# **Problem details**

Suppose we have a knapsack which can hold int w = 10 weight units. We have a total of int n = 4 items to choose from, whose values are represented by an array int[] val = {10, 40, 30, 50} and weights represented by an array int[] wt = {5, 4, 6, 3}.

Since this is the 0–1 knapsack problem, we can either include an item in our knapsack or exclude it, but not include a *fraction* of it, or include it *multiple* times.

# **Solution**

## **Step 1:**

First, we create a 2-dimensional array (i.e. a table) of n + 1 rows and w + 1 columns.

A row number *i* represents the set of all the items from rows 1— *i*. For instance, the values in row 3 assumes that we only have items 1, 2, and 3.

A column number *j* represents the weight capacity of our knapsack. Therefore, the values in column 5, for example, assumes that our knapsack can hold 5 weight units.

Putting everything together, an entry in row *i*, column *j* represents the maximum value that can be obtained with items 1, 2, 3 … *i*, in a knapsack that can hold *j* weight units.

Let’s call our table mat for matrix. Therefore, int[][] mat = new int[n + 1][w + 1].

## **Step 2:**

We can immediately begin filling *some* entries in our table: the base cases, for which the solution is trivial. For instance, at row 0, when we have *no items* to pick from, the maximum value that can be stored in any knapsack must be 0. Similarly, at column 0, for a knapsack which can hold 0 weight units, the maximum value that can be stored in it is 0. (We’re assuming that there are no massless, valuable items.)

We can do this with 2 for loops:

## **Step 3 (the crux of the problem):**

Now, we want to begin populating our table. As with all dynamic programming solutions, at each step, we will make use of our solutions to previous sub-problems.

**I’ll first describe the logic, before showing a concrete example.**

The relationship between the value at row *i*, column *j* and the values to the previous sub-problems is as follows:

Recall that at row *i* and column *j*, we are tackling a sub-problem consisting of items 1, 2, 3 … *i* witha knapsack of *j* capacity. There are 2 options at this point: we can either include item *i* or not. Therefore, we need to compare the maximum value that we can obtain with and without item *i*.

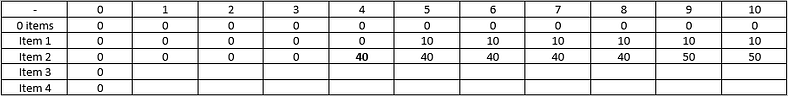
The maximum value that we can obtain *without* item *i* can be found at row *i-1*, column *j*. This part’s easy. The reasoning is straightforward: whatever maximum value we can obtain with items 1, 2, 3 … *i* must obviously be the same maximum value we can obtain with items 1, 2, 3 … *i - 1*, *if we choose not to include item i*.

To calculate the maximum value that we can obtain *with* item *i*, we first need to compare the weight of item *i* with the knapsack’s weight capacity. Obviously, if item *i* weighs more than what the knapsack can hold, we can’t include it, so it does not make sense to perform the calculation. In that case, the solution to this problem is simply the maximum value that we can obtain without item *i* (i.e. the value in the row above, at the same column).

However, suppose that item *i* weighs less than the knapsack’s capacity. We thus have the option to include it, if it potentially increases the maximum obtainable value. The maximum obtainable value by including item *i* is thus = the value of item *i* itself + the maximum value that can be obtained with the remaining capacity of the knapsack. We obviously want to make full use of the capacity of our knapsack, and not let any remaining capacity go to waste.

Therefore, at row *i* and column *j* (which represents the maximum value we can obtain there), **we would pick either the maximum value that we can obtain *without* item *i*, or the maximum value that we can obtain *with* item *i*, *whichever is larger*.**

Here’s a concrete example of what I mean.



At row 3 (item 2), and column 5 (knapsack capacity of 4), we can choose to either include item 2 (which weighs 4 units) or not. If we choose not to include it, the maximum value we can obtain is the same as if we only have item 1 to choose from (which is found in the row above, i.e. 0). If we want to include item 2, the maximum value we can obtain with item 2 is the value of item 2 (40) + the maximum value we can obtain with the remaining capacity (which is 0, because the knapsack is completely full already).

At row 3 (item 2), and column 10 (knapsack capacity of 9), again, we can choose to either include item 2 or not. If we choose not to, the maximum value we can obtain is the same as that in the row above at the same column, i.e. 10 (by including only item 1, which has a value of 10). If we include item 2, we have a remaining knapsack capacity of 9 - 4 = 5. We can find out the maximum value that can be obtained with a capacity of 5 by looking at the row above, at column 5. Thus, the maximum value we can obtain by including item 2 is 40 (the value of item 2) + 10 = 50.

We pick the larger of 50 vs 10, and so the maximum value we can obtain with items 1 and 2, with a knapsack capacity of 9, is 50.