no. of items	items	Bag Capacity (kg)				
available	available	0	1	2	3	4
0		0	0	0	0	0
1	A	0	0	5(A)	5(A)	5(A)
2	A,B	0	0	5(A)	6(B)	6(B)
3	A,B,C	0	0	5(A)	6(B)	7(C)
4	A,B,C,D	0	2(D)	5(A)	7(A,D)	8(B,D)

- We work through the table from top left corner to bottom right, going firstly from left to right across each row.
- The numbers in the body of the table are the *euro* values we can get, followed in brackets by the item choices.
- The number at the bottom right corner is the answer (8 euro: choose B,D)

To formalize this: let

PROF(i,j) ≡ Best profit we can obtain from choosing amongst the first i items, with bag capacity j.

With this definition:

- PROF(0,j) = 0 (there are no items to put in our bag).
- PROF(i,0) = 0 (our bag has capacity zero).

When we introduce a new item i (i.e. we go to a new row in the table above), we can either choose the new item or not:

Don't choose the new item i Then PROF(i,j) is just PROF(i-1,j) (i.e. our previous choice for best profit, not including new item i).

Choose the new item i From choosing the item i we get value v(i). But now the current bag capacity j is reduced to j - w(i). With this remaining bag capacity, the best profit we can obtain is $PROF(i-1, j-w(i))$. So our total best profit here is $v(i) + PROF(i-1, j-w(i))$.

Which of these two choices we make depends on which is best, i.e. we should choose the maximum! So we have

$$PROF(i,j) = MAX(PROF(i-1,j), v(i) + PROF(i-1, j-w(i)))$$

Note:

1. The Dynamic Programming Strategy here gives us all the intermediate results (if we wanted them!)
2. The way we order the items is immaterial to the final result, but affects the intermediate results.
3. What if our bag capacity (or item weights) were not integers? Then we need to make more "fine steps", in steps of e.g. 0.1, or 0.01 in the bag size.
4. The total complexity is just the size of the table (i.e. (n)(K)) so the algorithm is **O**(Kn). For example, for 20 items, bag capacity 50, we need to calculate (20)(50) numbers, i.e. 1,000 numbers. A Brute Force solution here requires $2^n = 2^{20} \approx 1,000,000$ calculations.

Addendum

The complexity quoted above for the Dynamic Programming solution to the Knapsack Problem *depends on how the solution is implemented*:

Iteration using a (2-dimensional) array We simply fill out the table above, from left to right and then from top to bottom: i.e.

- Calculate (write down!) PROF[0][0], PROF[0][1], PROF[0][2], ... (all these numbers are 0).
- Use the main equation to calculate PROF[1][0], PROF[1][1], PROF[1][2], ...
- Use the main equation to calculate PROF[2][0], PROF[2][1], PROF[2][2], ...
- etc.

- Use the main equation to calculate PROF[n][0], PROF[n][1], PROF[n][2], ... PROF[n][K]

Recursion We can just call PROF(i,j) as a *function*(of two variables i and j):

Base Case If either i or j is zero, return 0.

Recursive Step Use directly the main equation. (We must be careful here to use an `IF` statement to check that `w[i] < j`, otherwise we may end up with a negative index and hence infinite recursion.)

1. The initial call to PROF(n,K) leads to 2 new calls (specifically to PROF(n-1,K) and PROF(n-1, K-w[n]).
2. These then lead to 2 new calls each, a total of 4 calls to PROF(n-2,⁎).

3. This leads (in worst case) to 8 new calls to PROF(n-3,⁎).

In worst case this is $\mathbf{O}(2^n)$. In actual operation it will not be 2^n because in many cases (when w[i]>j: i.e. the item does not fit in the bag) the call to PROF(i-1, j-w[i]) will not be made. In the best case, w[i] is always greater than j (i.e. none of the items fit in to the bag!) and the problem is linear, $\mathbf{O}(n)$ with only n calls to the function.

So to summarize:

Small n, small K Doesn't matter which method you use!!

Small n, large K Use Recursion.

Large n, small K Use Iteration (2-dim array).

Large n, large K Use Iteration (2-dim array).

Nota Bene

When we say above, e.g. “small n, large K”, the comparison is not directly between n and K, rather between 2^n and K. For example, if $n = 20$ and $K = 1,000$, this might look like “small n, large K”, but when we compare 2^n to K,
$2^{10} = 1,048,576 \gg 1,000$, so here we should still use iteration:
K needs to be of the order of 2^n for recursion to win out.

Dynamic Programming Example 3: The Shortest Path Problem

We want to find the shortest path between any 2 cities on a (road) network. The distances of each “direct road” connection between neighboring cities are given.

Notation: we will write the distance between each of the n cities as an $n \times n$ matrix called A where $A(i, j)$ is the distance from city i to city j:

1. $A(i,j) = A(j,i)$, i.e. the matrix is symmetric (the distance from city i to city j is the same as the distance from city j to city i).
2. $A(i,i) = \infty \ \ \forall i$ (i.e. we don't want the traveller to go from a city to itself).
3. $A(i,j) = \infty$ if there is no direct road from i to j (i.e. if i and j are not neighboring cities).

Example

$$\begin{pmatrix} \infty & \infty & 2 & 12 \\ \infty & \infty & \infty & 3 \\ 2 & \infty & \infty & 5 \\ 12 & 3 & 5 & \infty \end{pmatrix}$$

The Dynamic Programming solution to this problem is commonly known as Floyd's Algorithm. We denote by

$$D_{ij}^{k} \equiv \text{The length of the shortest path from } i \text{ to } j \text{ where only interrmediate vertices with label less than or equal to } k \text{ are allowed.}$$

Note that

$$D_{ij}^{0} = A_{ij}.$$