Now let's look at an application of dynamic programming to a problem we have seen before – one of the most popular (and useful) optimization problems in the world:

0/1 Knapsack

Recall the problem definition. We have n objects $\{o_1, \ldots, o_n\}$, each with known mass m_i and known value v_i, and a container of capacity k. Our goal is to find a maximum-value subset of the objects that will fit in the container.

I'm going to present the dynamic programming solution very briefly - you should be able to fill in any gaps and complete the steps outlined above.

To identify subproblems that will be useful (and simultaneously see how to parameterize the problem) we can ask ourselves "what aspects of the problem definition can be reduced?" The most obvious answer is "k can be reduced" - and that is a good thought. Maybe there is some way to find the optimal solution for a container of size k by solving subproblems on a solutions of size < k. Let's hold on to the idea of using "container size" as one of the parameters.

Another aspect of the solution that can be reduced is the number of objects in the set. And here we encounter for the first time one of the most useful and powerful techniques for creating dynamic programming algorithms. For each object in the set there are really only two options: we either put it in the container or we don't. We can use these two options to build a recurrence relation in a very simple way.

Consider the last object in the set: o_n. If we put it in the container we get its value v_n, but the available capacity in which we can place other items is reduced by m_n. It we don't put it in the container we don't get its value but the available capacity is undiminished.

Let $MaxVal(i, t)$ be the maximum value we can get when the objects available to us are $\{o_1, \ldots o_i\}$ and the capacity is t.

Using this notation,

$$MaxVal(n,k) = max(v_n + MaxVal(n-1, k-m_n), \quad \text{\# use } o_n$$
$$MaxVal(n-1,k) \quad \text{\# don't use it}$$
$$)$$

and in fact we can use the same reasoning for o_{n-1} . We either include it or we don't, so if the objects available are $\{o_1 \ldots o_{n-1}\}$ and the available capacity is t, then

$$MaxVal(n-1,t) = max(v_{n-1} + MaxVal(n-2, t-m_{n-1}),$$
$$MaxVal(n-2,t)$$
$$)$$

This generalizes! If the objects available are $\{o_1, \ldots o_i\}$ then

$$MaxVal(i,t) = max(v_i + MaxVal(i-1, t-m_i),$$
$$MaxVal(i-1,t)$$
$$)$$

which works for all $i > 1$. $MaxVal(1,t)$ is the maximum value subset of $\{o_1\}$ that will fit in a container of capacity t. Clearly if $m_1 \leq t$ then the value is v_1, otherwise it is 0

$$MaxVal(1,t) = \begin{cases} v_1 & \text{if } m_1 \leq t \\ 0 & \text{if } m_1 > t \end{cases}$$

Those look like base cases, and they are … but we need one more generic base case to handle situations where $t < 0$ and we need one to handle the situation where $t = 0$

$MaxVal(i,t) = -\infty$ when $t < 0$

$MaxVal(i,t) = 0$ when $t = 0$

Now we have our recurrence relation complete, and it needed 2 parameters to cover all the situations.

Thus we can represent all the subproblem values using a 2-dimensional table. On one axis we will have all the objects, and on the other axis we place all the integers from 1 up to k. We can fill in the top row using our base cases for i=1. Then we can fill in each subsequent row. As explained above, each value depends on either 1 or 2 values from the row immediately above it. When completed, the bottom right hand element of the table gives the value of the optimal solution.

The key to understanding this application of dynamic programming is to see that if we know $MaxVal(i, t)$ for some i and for all t, then we can compute $MaxVal(i + 1, s)$ for all s. Furthermore, each of those computations takes constant time.

An example may help. Suppose we have 6 objects with these masses and values:

object	1	2	3	4	5	6
mass	7	4	6	4	3	5
value	100	70	40	80	25	80

And suppose k = 10

Our MaxVal table looks like this:

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1										
2										
3										
4										
5										
6										

We can fill in the first row using our base cases

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1	0	0	0	0	0	0	100	100	100	100
2										
3										
4										
5										
6										

And then the second row using values found in the first row.

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1	0	0	0	0	0	0	100	100	100	100
2	0	0	0	70	70	70	100	100	100	100
3										
4										
5										
6										

MaxVal(2,4) is the first point at which our recurrence relation becomes interesting. Recall $m_2 = 4$ and $v_2 = 70$

MaxVal(2,4) = max(70+MaxVal(1,0), MaxVal(1,4)) = max(70,0) = 70

When we get to MaxVal(2,7) we get

MaxVal(2,7) = max(70+MaxVal(1,3), MaxVal(1,7) = max(70,100) = 100

Continuing, the table looks like this:

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1	0	0	0	0	0	0	100	100	100	100
2	0	0	0	70	70	70	100	100	100	100
3	0	0	0	70	70	70	100	100	100	110
4	0	0	0	80	80	80	100	150	150	150
5	0	0	25	80	80	80	105	150	150	150
6	0	0	25	80	80	80	105	150	150	160

We see that the maximum possible value is 160. But how do we figure out which elements to choose?

We reconstruct the calculations that led to the 160:

MaxVal(6,10) = max(80+MaxVal(5,10-5), MaxVal(5,10))
= max(80+80,150)

which reveals that we achieved 160 by adding the 80 value of o_6 to the solution in the o_5 row with capacity = 10-5 = 5 . Thus our optimal solution includes object o_6

Where did the 80 in MaxVal(5,5) come from? We see that it came from the element directly above it (you should work out why this is true) which corresponds to the "do not include object o_5" decision. This element is MaxVal(4,5)

Using the same logic we discover that our optimal solution contains o_4 but none of the objects before that. Thus our optimal subset is $\{o_4, o_6\}$

Complexity

As we have seen, the MaxVal table has dimension n*k, and each element is computed in constant time. The "trace back" stage of the algorithm examines at most two elements on each row, and runs in O(n) time. Thus the whole algorithm runs in O(n*k + n) time, which simplifies to O(n*k) time.

But wait a second! We know that the 01-Knapsack problem is so hard that if it can be solved by a polynomial-time algorithm then P = NP. What's going on here?

It's a subtle point. $O(n * k)$ looks like it is a polynomial time class. But remember that k is part of the input – it is not fixed. Consider an instance where $k = 2^n$. For this instance our dynamic programming approach would be in $O(n * 2^n)$ which is certainly ***not*** polynomial.

Thus our algorithm cannot guarantee fixed-degree polynomial running time for all instances of the problem, so it does not prove P = NP. Good thing!