The 0-1 Knapsack Problem

The difference between this problem and the fractional one is that you can't take a fraction of an item. You either take the whole thing or none of it. So here, is the problem formally described:

Your goal is to maximize the value of a knapsack that can hold at most W units worth of goods from a list of items I_0, I_1, ... I_{n-1}. Each item has two attributes:

1) Value - let this be v_i for item I_i.
2) Weight - let this be w_i for item I_i.

Now, instead of being able to take a certain weight of an item, you can only either take the item or not take the item.

The naive way to solve this problem is to cycle through all 2^n subsets of the n items and pick the subset with a legal weight that maximizes the value of the knapsack. But, we can find a dynamic programming algorithm that will USUALLY do better than this brute force technique.

Our first attempt might be to characterize a sub-problem as follows:

Let S_k be the optimal subset of elements from $\{I_0, I_1,... I_k\}$. But what we find is that the optimal subset from the elements $\{I_0, I_1,... I_{k+1}\}$ may not correspond to the optimal subset of elements from $\{I_0, I_1,... I_k\}$ in any regular pattern. Basically, the solution to the optimization problem for S_{k+1} might NOT contain the optimal solution from problem S_k.

To illustrate this, consider the following example:

Item	Weight	Value
I_0	3	10
I_1	8	4
I_2	9	9
I_3	8	11

The maximum weight the knapsack can hold is 20.

The best set of items from $\{I_0, I_1, I_2\}$ is $\{I_0, I_1, I_2\}$ but the best set of items from $\{I_0, I_1, I_2, I_3\}$ is $\{I_0, I_2, I_3\}$. In this example, note that this optimal solution, $\{I_0, I_2, I_3\}$, does NOT build upon the previous optimal solution, $\{I_0, I_1, I_2\}$. (Instead it build's upon the solution, $\{I_0, I_2\}$, which is really the optimal subset of $\{I_0, I_1, I_2\}$ with weight 12 or less.)

So, now, we must rework our example. In particular, after trial and error we may come up with the following idea:

Let B[k, w] represent the maximum total value of a subset S_k with weight w. Our goal is to find B[n, W], where n is the total number of items and W is the maximal weight the knapsack can carry.

Using this definition, we have $B[0, w] = v_0$, if $w >= w_0$.
$= 0$, otherwise

Now, we can derive the following relationship that B[k, w] obeys:

$B[k, w] = B[k - 1,w]$, if $w_k > w$
$= \max \{ B[k - 1,w], B[k - 1,w - w_k] + v_k\}$

In English, here is what this is saying:

1) The maximum value of a knapsack with a subset of items from $\{I_0, I_1, ... I_k\}$ with weight w is the same as the maximum value of a knapsack with a subset of items from $\{I_0, I_1, ... I_{k-1}\}$ with weight w, if item k weighs greater than w.

Basically, you can NOT increase the value of your knapsack with weight w if the new item you are considering weighs more than w – because it WON'T fit!!!

2) The maximum value of a knapsack with a subset of items from $\{I_0, I_1, ... I_k\}$ with weight w could be the same as the maximum value of a knapsack with a subset of items from $\{I_1, I_2, ... I_{k-1}\}$ with weight w, if item k should not be added into the knapsack.

OR

3) The maximum value of a knapsack with a subset of items from $\{I_0, I_1, ... I_k\}$ with weight w could be the same as the maximum value of a knapsack with a subset of items from $\{I_0, I_1, ... I_{k-1}\}$ with weight $w-w_k$, plus item k.

You need to compare the values of knapsacks in both case 2 and 3 and take the maximal one.

Recursively, we will STILL have an $O(2^n)$ algorithm. But, using dynamic programming, we simply have to do a double loop - one loop running n times and the other loop running W times.

Question: In which cases would a running time of O(nW) be worse than a running time of $O(2^n)$?

Here is a dynamic programming algorithm to solve the 0-1 Knapsack problem. We will store our results in the array dp.

Input: S, a set of n items as described earlier, max the total weight of the knapsack. Assume that the weights and values are stored in separate arrays named weight and value, respectively.

Output: The maximal value of items in a valid knapsack. (The array dp will store the maximal values of all knapsacks up to weight max.

```
int[] dp = new int[max+1];
Arrays.fill(dp, 0);

for (int k=0; k<n; k++)
    for (int w = max; w>= weight[k]; w--)
        dp[w] = Math.max(dp[w], dp[w – weight[k]] + value[k])
```

Why is the inner loop running backwards?

Since our DP array isn't two dimensional (namely, instead of storing each row of the table, we store the last row and make the updates into the current row directly, we want to make sure that when we look in the dp array, we are not building off answers that were changed in the current loop iteration. By going backwards, we are only editing dp[w] based on dp[w-weight[k]] which has yet to be changed.

If we were to run the loop forwards, then if dp[w-weight[k]] was changed by using item k once, then dp[w] may use item k MORE THAN ONCE!!!

Thus, specifically, running the inner loop backwards prevents us from taking an item more than once.

Note on run time: Clearly the run time of this algorithm is O(nW), based on the nested loop structure and the simple operation inside of both loops. When comparing this with the previous $O(2^n)$, we find that depending on W, either the dynamic programming algorithm is more efficient or the brute force algorithm could be more efficient. (For example, for n=5, W=100000, brute force is preferable, but for n=30 and W=1000, the dynamic programming solution is preferable.)

Let's run through an example:

i	Item	w_i	v_i
0	I_0	4	6
1	I_1	2	4
2	I_2	3	5
3	I_3	1	3
4	I_4	6	9
5	I_5	4	7

W = 10

Item	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	6	6	6	6	6	6	6
1	0	0	4	4	6	6	10	10	10	10	10
2	0	0	4	5	6	9	10	11	11	15	15
3	0	3	4	7	8	9	12	13	14	15	18
4	0	3	4	7	8	9	12	13	14	16	18
5	0	3	4	7	8	10	12	14	15	16	19

How do we allow taking unlimited copies of an item?

Running the inner loop forwards will do the trick, since every previous item in the dp array has already been updated potentially using a copy of item k. This means that multiple copies of k can build up, if this is optimal.

The extra program Candy Store illustrates this idea.