Knapsack

Motivation: Imagine you are a robber, and break into a jewelry store. You can only carry out so much loot... what should you bring?

In this problem, you are given different items with weights and values, and a knapsack that can only hold a certain amount of weight. Your goal is to maximize the value of all the items in the knapsack without exceeding the total weight capacity of the knapsack.

Example: Knapsack capacity: 11. Item (which are (weight, value))

- Pearl: (2,11)
- Ring: (3, 19)
- Watch: (4, 25)

You could bring five Pearls, since that would have total weight 10, but not three Watches, since that would have total weight 12. Five Pearls would be worth 50, which is the same as two Watches. But maybe you should also grab some Rings...

This can be solved with dynamic programming. You create an array where each entry corresponds to the maximum value achieved by that weight. Then, to fill in the next entry, you see which item you could add to increase the value most, or add nothing. So, for example, for this problem at some point it would look like

Value	0	11	19	25	30	38	
Weight	1	2	3	4	5	6	7

So to get the best value for weight 6, you could add a Pearl, and the total value would be 39 ($10 + 29$, i.e. the value of the Pearl plus the best value for total weight 5), or you could add a Ring and the total value would be 44 ($19 + 25$, i.e. the value of the Ring plus the best value for total weight 4), or you could add a Watch and the total would be 44 ($25 + 19$, i.e. the value of the Watch plus the best value for total weight 3). So you add the Ring or Watch and continue on.

Value	0	11	19	25	30	38	44
Weight	1	2	3	4	5	6	7

Continue this process until you reach the weight of the knapsack.

Edit Distance

In this problem, you want to change one word into another word in as few edits as possible. Valid edits are adding a letter, deleting a letter, or changing a letter. So for example, to get from *shoes* to *socks*, you could (starting wtih *shoes*) delete the h, chance the e to a c, and add a k. That would be 3 edits.

Memoized Approach

It is easy to write a recursive algorithm for this problem. There are three possibilities: either we add a character, remove a character, or replace a character. The cost then is one plus the cost of computing the edit distance in the recursive call (unless we replace a character and the two are the same).

editDistance(s_1, s_2)
    If (length(s_1) == 0) **return** length(s_2)
    If (length(s_2) == 0) **return** length(s_1)
    c_1 = editDistance(chop(s_1), s_2) + 1
    c_2 = editDistance(s_1, chop(s_2)) + 1
    c_3 = editDistance(chop(s_1), chop(s_2))
    If (last(s_1) $\neq$ last(s_2))
        c_3 += 1
    return min(c_1, c_2, c_3)

However, like many recursive problems, we may end up re-doing a lot of work if we solve the same subproblem multiple times. Instead, we can set up a table to store the edit distances we have already computed and thus not have to recompute them. Whenever we compute an edit distance then, we insert it into the table, and before we make the recursive call we check the table to see if it's there. Here is the memoized version.

editDistance(s_1, s_2)
    If (length(s_1) == 0) **return** length(s_2)
    If (length(s_2) == 0) **return** length(s_1)
    If ((c_1 = table(chop(s_1), s_2)+1) == NULL)
        c_1 = editDistance(chop(s_1), s_2) + 1
    If (c_2 = table(s_1, chop(s_2)+1) == NULL)
        c_2 = editDistance(s_1, chop(s_2)) + 1
    If (c_2 = table(chop(s_1), chop(s_2)) == NULL)
        c_3 = editDistance(chop(s_1), chop(s_2))
    If (last(s_1) $\neq$ last(s_2))
        c_3 += 1
    distance = min(c_1, c_2, c_3)
    tableInsert(s_1, s_2, distance)
    return distance

Bottom-Up Approach

The other possibility is the bottom up approach. To do this, you set up a table where columns represent the letters of s_1 and rows represent the letters of s_2, and you fill in the table from left to right, top to bottom. At each cell, you have three choices, corresponding to the previous recursive choices. Deleting a letter from s_1 means you look at the distance above you to determine the smaller edit distance. Inserting a letter to s_1 means you look at the distance to your left. Replacing one

with the other means you look at the distance to your upper left. The bottom right corner at the end has the edit distance of the two words.