# An introduction to dynamic programming

Below are two examples of problems that can be solved with dynamic programming. This algorithmic paradigm is quite powerful and appears in many settings (e.g., Dijkstra's algorithm for computing the shortest path in a graph is a variant of dynamic programming).

At a very high level, dynamic programming tries to decompose the problem you are solving so that you solve increasingly simpler problems until you reach a base case. The key is to define a set of **recurrence equations** which define all the possible ways of going from a more complex to a simpler problem. At some level, dynamic programming is similar to solving a problem in a recursive way, however dynamic programming ensures you do not duplicate effort, which is not necessarily true in the recursive solution. All of this will be more clear as we go through the following problems.

## Making change

You should be quite familiar with trying to make change for a particular dollar amount. We usually don't worry too much about the number of coins or bills that we have to use to reach a certain amount, however, imagine you want to use the minimum number possible of pieces of currency. How would you determine that number (and the actual value of the currency involved)?

To formalize this a bit better, assume you have a collection of coins with *d* distinct values V=[$v_1$, $v_2$, ..., $v_d$], and you are trying to make change for a total dollar value of *D*. In other words, $D = \sum_i n_i v_i$ where $n_i$ is the number of coins of denomination *i* that you used (n*i* can be *0* if that denomination was not used). We want to find the set of $n_i$ values such that $\sum_i n_i$ is minimized over all possible choices of values that add up to *D*.

A common heuristic for solving this problem follows a greedy approach where you use the highest denomination coins that add up to less than D, then continue until you reach the total amount. For some sets of denominations, this strategy also yields an optimal solution, however this is not true in general. The textbook proposes a Roman currency scheme that had *V=[120, 40, 30, 24, 20, 10, 5, 4, 1]*, and showed that for the value *D=48*, the greedy scheme fails to find the optimal solution.

Thus, how do you think about this problem? To start with, let us define the problem we are solving as a function *N(D)* which represents the minimum number of coins needed to come up with the amount *D*.

Now that we have this definition, we can try to figure out how to recursively reduce the complexity of the problem. So, let's focus on just one coin. The optimal solution could use one coin of any denomination. We don't yet know which one, but we know that once we pick one coin, the problem becomes simpler.

For example, assume you pick a coin of value $v_1$, the minimum number of coins you need to make up up the amount *D* is going to be *1* larger than the minimum number of coins you need to make up $(D-v_1)$ , which is a simpler problem. Note that you don't really know if the optimal solution uses a coin of value $v_1$, but you know that the optimal solution has to use at least one coin of some denomination, so

you simply try them all to see which one gives you a better solution (fewer coins). Specifically, you can say that:

$$N(D) = min \begin{cases} N(D - v_1) + 1 \\ N(D - v_2) + 1 \\ \cdots \\ N(D - v_d) + 1 \end{cases}$$

You can write this as a recursive function in your favorite programming language. One bit that is missing is the base case/stopping condition, but that should be easy to figure out. *N(D) = 0* if *D < min(V)* (trying to make change for an amount smaller than the smallest coin).

Note, however, that this recursive definition can be quite wasteful as you may call the function N with the same value D multiple times. For example, assume V contains values 2, 4, and 8. In the computation of N(D) you will have to figure out the value for N(D – 8). But you will also reach this number when trying to compute N(D – 4), which then will need to know the value of N(D – 4 – 4) = N(D – 8). Likewise N(D – 2) will eventually lead to N(D – 2 – 2 – 2 – 2) = N(D – 8).

This is where dynamic programming comes in handy by storing the N(D) values when they are first computed, then referring to them whenever we need them again, without having to recompute them. To do so, we flip the problem around, rather than going from a more complex to a simpler problem, we fill in the table going from the simpler problems to the more complex using the recurrence information.

Assume, thus, that we have the array *N[]* that contains the values we want to compute. *N[D]* will eventually contain the minimum number of coins that can be used to make change for D. We'll initialize *N[0]* to 0 (you don't need any coins to make change for $0), and the other entries in N to be $+\infty$. Then we can fill the matrix using the following approach:

```
for i = 0, D - 1
  for j in coin values
     if i + j > D
        continue to next j
     if N[i] + 1 < N[i + j]  # pick smaller number of coins
        N[i + j] = N[i] + 1
```

Essentially, you are adding one coin to the amount you currently have and see if that gives you a smaller number of coins for the new value. When you reach N[D] the value in the matrix will have the smallest number of coins needed to reach D.

# Longest common subsequence

In the case of counting change, the dynamic programming solution was one-dimensional, breaking up the problem into a series of D sub-problems, where D was the target dollar amount. Here we present a problem that yields a two-dimensional dynamic programming solution.

The Longest Common Subsequence problem seeks to find the longest set of letters that is shared in the same order in two strings. For example, for strings PLEASANTLY and MEANLY, a possible solution

is the string EALY as the letters E, A, L, and Y all occur in the same order in the two strings, highlighted in underlined bold below:

PL**EA**SANT**LY**, M**EA**N**LY**

Let's try to solve this problem using a different logic as we did with the coin problem.

First, we'll define a function LCS that takes in the two strings as parameters and returns the length of the longest subsequence. While we don't know how to solve this problem for the full strings, we can already figure out a base case: if either string is empty, the length of the longest common subsequence is 0. But how can we transform this complex problem to something simpler, that will recursively lead to the correct answer?

An obvious (perhaps) definition of "simpler" is a version of LCS applied to (slightly) shorter strings. To make things easier to wrap our minds around, let's look at how we can relate the longer strings provided as input, to strings that are just one character shorter. Can we relate LCS(PLEASANTLY, MEANLY) to LCS(PLEASANTL, MEANLY), or LCS(PLEASANTL, MEANL)?

For these particular strings, we can observe that the last characters match each other, thus they are a plausible contribution to the LCS values, thus we can say that:

LCS(PLEASANTLY, MEANLY) = 1 + LCS(PLEASANTL, MEANL)

Note, however that this happens to only be the case for these particular strings. What if the strings were PLEASANTLY, MEANYL? By deleting the last character of both strings we will not be able to match either the Y or the L characters:

PLEASANTL
MEANY

Thus, perhaps we may miss an optimal solution. However it is also possible that this is indeed the right choice and:

LCS(PLEASANTLY, MEANYL) = 0 + LCS(PLEASANTL, MEANY)

To avoid missing possible character matches, we can try to remove one character from one or the other string, looking at:

LCS(PLEASANTLY, MEANY) and LCS(PLEASANTL, MEANYL)

In both cases, you can see that the last characters match, thus we can discover these possible matches and extend the LCS value.

At this point, however, we have three possible choices for LCS(PLEASANLY, MEANYL):

0 + LCS(PLEASANTL, MEANY)
0 + LCS(PLEASANTLY, MEANY)
0 + LCS(PLEASANTL, MEANYL)

In fact, there are only three ways to make the strings shorter by one character, thus these are the only choices available to us. In all the cases here, we have added 0 because the characters removed from the

strings do not match each other, thus cannot contribute to the LCS. Which one of these choices should we select?

Since we are trying to find the **longest** common subsequence, we will choose the version that gives us the highest value, or:

$$LCS(PLEASANTLY, MEANYL) = max\begin{cases} LCS(PLEASANTL, MEANY) \\ LCS(PLEASANTLY, MEANY) \\ LCS(PLEASANTL, MEANYL) \end{cases}$$

Ok, this is a bit of a handful to write. so we'll use the shortcut we used for suffix arrays, and just refer to the position where the string ends, knowing we can always figure out the actual characters within the original string. Thus, let's define:

S1 = PLEASANTLY
S2 = MEANYL

The last characters in the two strings are, respectively, S1[9] and S2[5], thus we can write the equation above as:

$$LCS(9,5) = max\begin{cases} LCS(8,4) \\ LCS(9,4) \\ LCS(8,5) \end{cases}$$

More generally, if the strings S1 and S2 have lengths n and m, respectively, we can say that:

$$LCS(n,m) = max\begin{cases} LCS(n-1,m-1) \\ LCS(n,m-1) \\ LCS(n-1m) \end{cases}$$

But, remember, that we added 0 to LCS(n-1, m-1) because we knew that the last letters in PLEASANTLY and MEANYL did not match. In the case of PLEASANTLY and MEANLY the last letters matched and we added 1. Thus, for the general case, we have to add a conditional test:

$$LCS(n,m) = max\begin{cases} LCS(n-1,m-1) + 1(if\ S1[n] = S2[m]) \\ LCS(n,m-1) \\ LCS(n-1m) \end{cases}$$

Just like we did for the coin change problem, we can now apply this function recursively to solve the LCS problem, taking into account the base cases LCS(-1, m) = LCS(n, -1) = 0 (empty strings don't contribute to LCS.

Note that we can convert this to a dynamic programming matrix, which is now two-dimensional, corresponding to the two strings:

| | - | 0 | 1 | ... | n |
|---|---|---|---|---|---|
| - | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | | | | |
| 1 | 0 | | | | |
| ... | 0 | | | | |
| m | 0 | | | | |

Note that we have included characters '-' to represent the empty string, thus encoding the base case in the matrix. The cells of the matrix can be filled in following the recurrence discussed above, with cell LCS[i, j] depending on the three adjacent cells LCS[i – 1, j – 1], LCS[i, j – 1], and LCS[i – 1, j], corresponding to the three cases we discussed.

As an example, here is the matrix for the two strings we were working with above:

| | - | P | L | E | A | S | A | N | T | L | Y |
|---|---|---|---|---|---|---|---|---|---|---|---|
| - | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| M | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| E | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| A | 0 | 0 | 0 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| N | 0 | 0 | 0 | 1 | 2 | 2 | 2 | 3 | 3 | 3 | 3 |
| Y | 0 | 0 | 0 | 1 | 2 | 2 | 2 | 3 | 3 | 3 | 3 |
| L | 0 | 0 | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 4 | 4 |

Thus, we can tell that the length of the longest common subsequence of the two strings is 4. Figuring out which characters actually represent the longest common subsequence, we need to backtrack from the bottom right corner, following the path that led to the value recorded in each cell, until we reach the top left corner. More details on this process are provided in the description of the sequence alignment algorithm.