

Practice Problems on Dynamic Programming

September 11, 2001

Below are two practice problems on designing and proving the correctness of dynamic programming algorithms. For those of you who feel like you need us to guide you through some additional problems (that you first try to solve on your own), these problems will serve that purpose. *If anyone would like a help session where I guide you through the process of solving these problems, please let me know.*

The front page has the problems and the rest gives the solutions. You can use these solutions as a guide as to how you should write-up your solutions. These problems will be MUCH more valuable to you if you first solve them and then check the solutions.

Practice Problems

1. Suppose we want to make change for n cents, using the least number of coins of denominations 1, 10, and 25 cents.

Describe an $O(n)$ dynamic programming algorithm to find an optimal solution.

2. Here we look at a problem from computational biology. You can think of a DNA sequence as sequence of the characters "a", "c", "g", "t". Suppose you are given DNA sequences D_1 of n_1 characters and DNA sequence D_2 of n_2 characters. You might want to know if these sequences appear to be from the same object. However, in obtaining the sequences, laboratory errors could cause reversed, repeated or missing characters. This leads to the following sequence alignment problem.

An alignment is defined by inserting any number of spaces in D_1 and D_2 so that the resulting strings D'_1 and D'_2 both have the same length (with the spaces included as part of the sequence). Each character of D'_1 (including each space as a character) has a corresponding character (matching or non-matching) in the same position in D'_2 . For a particular alignment A we say $cost(A)$ is the number of mismatches (where you can think of a space as just another character and hence a space matches a space but does not match one of the other 4 characters).

To be sure this problem is clear suppose that D_1 is `ctatg` and D_2 is `ttaagc`. One possible alignment is given by:

```
ct at g
  tta agc
```

In the above both D'_1 and D'_2 have length 8. The cost is 5. (There are mismatches in position 1, 3, 5, 6 and 8).

Give the most efficient algorithm you can (analyzed as a function of n_1 and n_2) to compute the alignment of minimum cost.

Solutions (solve the problems before reading this)

- Below is a dynamic programming solution for this problem to illustrate how it can be used. There is a very straightforward $O(1)$ time solution. It can be shown that if $n \geq 50$ then any solution will include a set of coins that adds to exactly 50 cents. Hence it can be shown that an optimal solution uses $2 \cdot \lfloor n/50 \rfloor$ quarters along with an optimal solution for making $n/50 - \lfloor n/50 \rfloor$ cents which can be looked up in a table of size 50.

Here's the dynamic programming solution for this problem. (It does not use the fact that an optimal solution can be proven to use $2 \cdot \lfloor n/50 \rfloor$ quarters and hence is not as efficient.) The general subproblem will be to make change for i cents for $1 \leq i \leq n$. Let $c[i]$ denote the fewest coins needed to make i cents. Then we can define $c[i]$ recursively by:

$$c[i] = \begin{cases} \text{use } i \text{ pennies} & \text{if } 1 \leq i < 9 \\ c[i - 10] + 1 & \text{if } 10 \leq i < 24 \\ \min(c[i - 10] + 1, c[i - 25] + 1) & \text{if } i \geq 25 \end{cases}$$

Note that $c[n]$ is the optimal number of coins needed for the original problem.

Clearly when $i < 10$ the optimal solution can use only pennies (since that is the only coin available). Otherwise, the optimal solution either uses a dime or a quarter and both of these are considered. Finally, the optimal substructure property clearly holds since the final solution just adds one coin to the subproblem solution. There are n subproblems each of which takes $O(1)$ time to solve and hence the overall time complexity is $O(n)$.

- Recall that D_1 is a DNA sequence with n_1 characters and D_2 is a DNA sequence with n_2 characters. The general form of the subproblem we solve will be: Find the best alignment for the first i characters of D_1 and the first j characters of D_2 for $1 \leq i \leq n_1$ and $1 \leq j \leq n_2$. Let $D(i)$ be the i th character in string D . Let $c[i, j]$ be the cost of an optimal alignment for $D_1(1), \dots, D_1(i)$ and $D_2(1), \dots, D_2(j)$. We can define $c[i, j]$ recursively as shown (where $c[n_1, n_2]$ gives the optimal cost to the original problem):

$$c[i, j] = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ c[i - 1, j - 1] & \text{if } D_1(i) = D_2(j) \\ \min\{c[i - 1, j - 1], c[i - 1, j], c[i, j - 1]\} + 1 & \text{otherwise} \end{cases}$$

We now argue this recursive definition is correct. You can form D'_1 and D'_2 (and hence the alignment) for the subproblem from the right to left as follows. In an optimal alignment either the last character of D'_1 is a space or it is the last character (character i) of D_1 and the last character of D'_2 is a space or it is the last character (character j) of D_2 . If $D_1(i) = D_2(j)$ then clearly it is best to align them (so add a space to neither). However, if $D_1(i) \neq D_2(j)$ then a space could be added to neither or just one. In all three cases one mismatch is caused by the last characters. Notice that there would never be any benefit in ending both D_1 and D_2 with a space. Hence the above recursive definition considers all possible cases that the optimal alignment could

have. Since the solution to the original problem is either the value of the subproblem solution (if $D_1(i) = D_2(j)$) or otherwise one plus the subproblem solution, the optimal substructure property clearly holds. Thus the solution output is correct.

For the time complexity it is clearly $O(n_1 \cdot n_2)$ since there are $n_1 \cdot n_2$ subproblems each of which is solved in constant time. Finally, the $c[i, j]$ matrix can be computed in row major order and just as in the LCS problem a second matrix that contains which of the above 4 cases was applied can also be stored and then used to construct an optimal alignment.