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
   ttaagc

   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)

1. 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 $2 \cdot [n/50]$ quarters along with an optimal solution for making $n/50 - [n/50]$ 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 $2 \cdot [n/50]$ quarters and hence is not as efficient.) The general subproblem will be to make change for $n$ optimal solution for making 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

Here's the dynamic programming solution for this problem. (It does not use the fact that an optimal solution can be proven to use $2 \cdot [n/50]$ quarters and hence is not as efficient.) The general subproblem will be to make change for $n$ optimal solution for making 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.