Below is a set of practice problems on designing and proving the correctness of greedy algorithms, to help you check your understanding of the approach we’ve discussed in class.

You can find solutions to these problems on the course web site. We’re also willing to listen to you describe your solutions or to look at your writeups during office hours.

Note: If you want to discuss these problems, I expect you to give correctness and complexity proofs for your algorithms here, just as on the actual homework. Efficiency matters!

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.

   Give an \( O(n) \) dynamic programming algorithm to find the optimal set of change. (Yes, there’s a way to make this \( O(1) \), but see if you can work through the dynamic programming algorithm first.)

2. You are driving from New York to San Francisco, using one or more rental cars for the trip. Along the way, you will visit cities \( c_1 \ldots c_n \) in order (here, \( c_1 \) is New York and \( c_n \) is San Francisco). You have a fee schedule \( f(i, j) \) that gives the cost of a rental that is picked up in city \( c_i \) and dropped off in city \( c_j \), \( j > i \). Note that these costs are arbitrary and possibly non-monotonic; for example, it may cost $200 for a rental from \( c_1 \) to \( c_2 \) but only $100 for a rental from \( c_1 \) to \( c_3 \).

   Give an algorithm to choose a set of rentals that minimize the total cost of your trip. Note that you can rent only one car at a time, and that you can never be without a car. Hence, a solution is a set of non-overlapping rentals that span all cities.

3. A well-known problem, related to longest common subsequence, for which dynamic programming works is string edit distance. You are given strings \( S \) and \( T \) (not necessarily of the same length) over a common alphabet \( \Sigma \). The goal is to determine the minimum number of edit operations needed to transform \( S \) into \( T \). The valid edit operations are:

   (a) substitution: change one character of \( S \) to a new one
   (b) insertion: add a character at an arbitrary position in \( S \)
   (c) deletion: remove a character from an arbitrary position in \( S \)

   For example, transforming the string “acaat” into “atagaa” can be done with three edits, as shown by the following alignment of the two strings:

   \[
   \begin{align*}
   &aca-aat \\
   &atagaa-
   \end{align*}
   \]

   Here, a gap “—” in the top sequence indicates that the character in the bottom sequence was inserted, while a gap in the bottom sequence indicates that the character in the top sequence was deleted.
Give an efficient algorithm to compute the edit distance between strings $S$ and $T$ of lengths $n$ and $m$. You may want to think of this problem as finding an alignment that realizes this edit distance.

(Note: this computation is the core of the famous Smith-Waterman algorithm for determining the similarity of DNA and protein sequences.)