WARNING: if you haven’t at least tried hard to solve the practice problems before reading these solutions, you are missing the point. If you can’t make any progress, talk to me or to the TAs before reading these solutions. Otherwise, you should come up with a solution of your own that you can compare to the one shown here.

Note: there is usually more than one way to solve a dynamic programming problem; for example, you may go forward, backward, or in some other order entirely. As long as you can check that your solution is OK, it doesn’t matter if you use a different problem decomposition than the one shown here. If you came up with a solution that is still polynomial-time but slower than mine, look carefully to see why mine is faster.

1. When changing \(n\) cents, there are 3 choices we can make for the first coin: penny, dime, or quarter.

**Complete Choice Property:** At least one of these choices leads to an optimal solution, since one of these coins must be in any solution.

(Strictly speaking, this is true only if \(n \geq 25\); for \(10 \leq n \leq 24\), we can use only a penny or a dime, and for \(n < 10\), we can use only a penny. We’ll deal with the general case in this proof and leave the others as an exercise.)

**Inductive Structure:** Each initial choice leaves us with a smaller subproblem. Indeed, after choosing any one coin \(c_k\) with value \(k\) cents, we are left to make change for the remaining \(n - k\) cents. Any way of making change in the subproblem is compatible with the initial choice.

**Optimal Substructure:** Let \(P\) be the original problem of changing \(n\) cents, and suppose our first choice is a coin \(c_k\) worth \(k\) cents. Let \(\Pi'_k\) be an optimal solution to the subproblem of changing \(n - k\) cents. Then \(\Pi_k = \Pi'_k \cup \{c_k\}\) is optimal among all solutions to \(P\) that start with \(c_k\).

**Pf:** the value of the overall solution \(\Pi_k\) for any choice \(c_k\) is given by

\[
\text{value}(\Pi_k) = \text{value}(\Pi'_k) + 1.
\]

Apply the usual contradiction proof.

We now develop a recurrence for this problem. Subproblems may be indexed by the number \(j\) of cents for which we need to make change, which implies a one-dimensional index \([j]\) for \(1 \leq j \leq n\).

Let \(C[j]\) be the cost of an optimal solution for \(j\) cents, and let \(C[j \mid k]\) be the cost of an optimal solution given that the first coin chosen is \(c_k\) with value \(k\) cents. Then we have that

\[
C[j] = \min (C[j \mid 1], C[j \mid 10], C[j \mid 25])
\]

where for each denomination \(k\),

\[
C[j \mid k] = 1 + C[j - k].
\]

The base cases are, strictly speaking, precomputed optimal values for all \(j < 25\); alternatively, when \(j < 25\), we choose between only the first two options, and when \(j < 10\), \(C[j] = j\). The goal point is \([n]\), the original input value for the problem instance.

To compute this recurrence bottom-up, we may compute \(C[j]\) for \(1 \leq j \leq n\), since \(C[j]\) depends only on the costs of strictly smaller subproblems. Each \(C[j]\) is computed as a maximum over three options in time \(\Theta(1)\), and the domain size is \(n\), so the overall algorithm is \(\Theta(n)\).
2. We’ll solve this problem by working backward from the end of the trip. The last rental in the chain
takes us from some city \( c_i \) to the final city \( c_n \); denote this rental by \( (c_i, c_n) \).

**Complete Choice Property:** Each possible index \( 1 \leq i < n \) for the last city before \( c_n \) is a choice. An optimal solution must choose *some* \( c_i \), \( i < n \), at which to begin the last rental, so considering all of them guarantees that we will consider a choice consistent with an optimum.

**Inductive Structure:** once we choose a starting city \( c_i \) for the last rental, we are left with the smaller subproblem of getting from \( c_i \) to \( c_j \). Any strategy that gets us to \( c_i \) is consistent with the initial choice, since it yields a feasible overall plan for getting from \( c_i \) to \( c_n \).

**Optimal Substructure:** Let \( \Pi_i' \) be an optimal solution to the subproblem of getting from \( c_i \) to \( c_i \). Then for all \( i \), the solution \( \Pi_i = \Pi_i' \cup (c_i, c_n) \) to the original problem instance is optimal among all solutions whose last rental starts at \( c_i \).

**Pf:** the cost of the solution \( \Pi_i \) is given by

\[
\text{cost}(\Pi_i) = \text{cost}(\Pi_i') + f(i, n).
\]

Apply the usual contradiction argument.

We now develop a recurrence for this problem. We must consider the subproblems of getting from \( c_1 \) to each possible \( c_i \). We can index these subproblems by the one-dimensional index \([i]\).

Let \( C[i] \) be the cost of an optimal solution for getting from \( c_1 \) to \( c_i \), and let \( C[i \mid k] \) be the cost of an optimal solution given that the last step is \((c_k, c_i)\). We have that

\[
C[i] = \min_{k<i} C[i \mid k]
\]

where

\[
C[i \mid k] = C[k] + f(k, i),
\]

Our base case is that \( C[1] = 0 \), since we start from \( c_1 \) and need no rental to get there. Our goal point is \([n]\), the problem of getting from \( c_1 \) to \( c_n \).

To compute this recurrence bottom-up, we may compute \( C[i] \) in increasing order from 1 to \( n \). This order satisfies all dependencies because each subproblem \([i]\) depends only on earlier subproblems. There are \( n \) points in the domain, and computing \( C[i] \) minimizes over \( \Theta(i) \) previously computed values, so the entire algorithm takes \( \Theta(n^2) \) total time.

3. The goal of this problem is to find an alignment between strings \( S \) and \( T \) that requires the fewest possible edits. We’ll build up this alignment working backwards from the end of the two strings. Consider the last position of the alignment between \( S \) and \( T \). There are three possibilities for the content of this position. First, we may align \( S[n] \) to \( T[m] \). If these characters match, there is no edit at this position; else, there is a substitution. Second, we may align \( S[n] \) to a gap, postulating deletion of \( S[n] \). Third, we may align \( T[m] \) to a gap, postulating insertion of \( T[m] \).

**Complete Choice Property:** an optimal alignment of \( S \) and \( T \) either matches its last two characters or aligns at least one to a gap, so the set of choices is consistent with optimality. (Note: an optimal solution never aligns both of the last two characters to a gap, but that is not important to the proof.)

**Inductive Structure:** for each choice of the last alignment position, we are left with a subproblem of finding an optimal alignment on substrings of \( S \) and \( T \). If \( S[n] \) aligns to \( T[m] \), we must align \( S[1..n-1] \) to \( T[1..m-1] \). If \( S[n] \) aligns to a gap, we must align \( S[1..n-1] \) to \( T[1..m] \). If \( T[m] \) aligns to a gap, we must align \( S[1..n] \) to \( T[1..m-1] \).
All of these subproblems are without external constraints, since in each case, any alignment of the remaining prefixes is consistent with the corresponding alignment of the last character(s).

**Optimal Substructure:** For each of the three choices, suppose we solve the subproblem optimally. Then we get an optimal solution to the full problem, given that choice.

**Pf:** Let $\delta(x, y)$ be 0 if $x = y$ and 1 otherwise. Then, letting the score of an alignment be the number of edits it induces, we have the following three observations:

(a) If $S[n]$ aligns to $T[m]$, then

$$\text{score}(S[1..n], T[1..m]) = \text{score}(S[1..n-1], T[1..m-1]) + \delta(S[n], T[m]).$$

(b) If $S[n]$ aligns to a gap, then

$$\text{score}(S[1..n], T[1..m]) = \text{score}(S[1..n-1], T[1..m]) + 1.$$  

(c) If $T[m]$ aligns to a gap, then

$$\text{score}(S[1..n], T[1..m]) = \text{score}(S[1..n], T[1..m-1]) + 1.$$  

Apply the standard contradiction argument to all three cases.

We now construct the recurrence. The domain of subproblems can be indexed by $[i,j]$, corresponding to the problem of aligning $S[1..i]$ with $T[1..j]$.

Let $\sigma(i, j)$ be the score of an optimal alignment of $S[1..i]$ with $T[1..j]$. Then (skipping the conditional notation), our optimal substructure proof tells us that

$$\sigma(i, j) = \min \left\{ \begin{array}{ll} \sigma(n-1, m-1) + \delta(S[n], T[m]) \\ \sigma(n-1, m) + 1 \\ \sigma(n, m-1) + 1 \end{array} \right\}$$

Our base cases are $\sigma(i, 0) = i$ and $\sigma(0, j) = j$, since these cases represent deletion of $i$ and insertion of $j$ characters, respectively. Our goal is to compute $\sigma(n, m)$.

To solve the problem bottom-up, we can (among other orderings) compute $\sigma(i, j)$ in order first by $i$, then by $j$. This ordering satisfies the data dependencies, as discussed in the previous problem. The computation cost is $\Theta(1)$ per point $[i, j]$, and there are $\Theta(mn)$ points in the domain, so the total algorithm cost is $\Theta(mn)$. 
