1. The following algorithm is used in spelling checkers and correctors, as well as in determining string similarity in genetics research. You are given two strings, \( X = <x_1 \ldots x_m> \) and \( Y = <y_1 \ldots y_n> \). Define the edit distance between \( X \) and \( Y \) to be the minimum number of single character insertions, deletions, and single-character replacements that are applied to \( X \) to make it equal to \( Y \). For example, if \( X = <\text{barney}> \) and \( Y = <\text{crony}> \) then the edit distance is 4. The sequence of changes is:

   - Replace \( x_1 \) (‘b’) with \( y_1 \) (‘c’).
   - Delete \( x_2 \) (‘a’).
   - Insert \( y_3 \) (‘o’) after \( x_3 \) (‘r’).
   - Delete \( x_5 \) (‘e’).

The objective of this problem is to derive an \( O(mn) \) time and \( O(mn) \) space algorithm for this problem.

a. Give a dynamic programming formulation (the recursive rule) for determining the minimum edit distance between \( X \) and \( Y \).

b. Use your recursive rule to generate pseudo-code for a dynamic programming algorithm for this problem. Your algorithm should also contain the necessary “hooks” so that you can extract the actual edit operations (see part (c)).

c. Using the algorithm from (b), give pseudo-code for an algorithm that outputs the sequence of editing operations (in a form similar to the one given above).
2. The objective of this problem is to write a dynamic programming algorithm to play a game. Two players, called Jen and Ben alternate in taking moves, with Jen always going first. Initially the board consists of three piles of diamonds, which we denote (A, B, C), meaning that there are A diamonds in the first pile, B in the second pile, and C in the third. The board always consists of three numbers that are nonnegative. During a move a player can do any one of the following:

1. Remove 1 diamond from pile 1.
2. Remove either 1 or 2 diamonds from pile 2.
3. Remove either 2 or 3 diamonds from pile 3.

The first player who cannot make a move loses. (And the winner gets all the diamonds.) That is, if it is a player’s turn to move and the board is either (0, 0, 0) or (0, 0, 1) then he/she loses. Given the initial configuration, (A,B,C), and with the assumptions that Jen plays first and both players play as well as possible, determine which of the two players can force a win. (Since there is no element of chance, and the game is finite in length, one of the two can always force a win.)

For example, suppose that the initial board is (0, 1, 4). In this case Jen can force a win. She uses rule (3) to remove 2 diamonds from pile 3, resulting in the board (0, 1, 2). Ben’s only choices are to remove 1 from pile 2 or 2 from pile 3, resulting in the possible boards (0, 0, 2) and (0, 1, 0). In either case, Jen can remove the final diamonds (using either rules (3) or (2), respectively) leaving Ben without a move.

a. Derive a (recursive) dynamic programming rule to determine the winner, given the initial board (A, B, C). (Be sure to include a description of the basis cases.) Justify the correctness of your formulation. (For this part I do not want a full algorithm, just the recursive rule.)

b. Present an implementation of recursive rule of part (a). (You may use memoization or the bottom-up method.) Express your running time as a function of A, B, and C.

3. Describe a polynomial time algorithm for the following problem. You are given a flow network G = (V, E) with source s and sink t, and whose edges all have unit capacity, that is, c(u, v) = 1 for all (u, v) ∈ E. Your algorithm is also given a parameter k, where 1 ≤ k ≤ m, where m = |E|. The objective is to delete k edges from G so as to reduce the maximum s-t flow value as much as possible. In other words, compute a subset E′ ⊆ E of size k such that the flow in G′ = (V,E \ E′) (E \ E′ means E not in E′) is as low as possible.

Justify your algorithm’s correctness. (Hint: Your solution may involve computing a network flow in G or some variant of G.) Let T_f (n,m) denote the running time of a good algorithm for network flow on a network with n vertices and m edges. Express the running time of your algorithm as a function of n, m, and T_f (n,m).
4. Network flow issues come up in dealing with natural disasters and other crises, since major unexpected events often require the movement and evacuation of large numbers of people in a short amount of time. Consider the following scenario. Due to large-scale flooding in a region, paramedics have identified a set of $n$ injured people distributed across the region who need to be rushed to hospitals. There are $k$ hospitals in the region, and each of the $n$ people needs to be brought to a hospital that is within a half-hour’s driving time of their current location (so different people will have different options for hospitals, depending on where they are right now). At the same time, one doesn’t want to overload any one of the hospitals by sending too many patients its way. The paramedics are in touch by cell phone, and they want to collectively work out whether they can choose a hospital for each of the injured people in such a way that the load on the hospitals is balanced: Each hospital receives at most $\lceil n/k \rceil$ people. Give a polynomial-time algorithm that takes the given information about the people’s locations and determines whether this is possible.

5. **Programming Problem:** Consider the shortest common supersequence (SCS) problem. The input to the SCS problem is two strings $A = A_1 \ldots A_m$ and $B = B_1 \ldots B_n$, and the output is the length of the shortest string that contains both $A$ and $B$ as subsequences.

**Example:** $A = < ABCBABA>$, $B = < BCAABAB>$, then the $SCS = < ABCAABABA>$, so the $\text{len}(SCS) = 9$.

Given that the recurrence relationship is as follows, implement a dynamic programming algorithm to solve the SCS problem. Let $SCS(i, j)$ be the length of the shortest common supersequence of $A[1..i]$ and $B[1..j]$. Your algorithm must run in $O(mn)$ time.

Your program should prompt the user to enter two strings and output the length of the shortest common supersequence, not the actual SCS.

\[
SCS(i, j) = \begin{cases} 
  j & \text{if } i = 0 \\
  i & \text{if } j = 0 \\
  SCS(i - 1, j - 1) + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\
  SCS(i, j - 1) + 1 & \text{if } i, j > 0 \text{ and } x_i \neq y_j \\
  SCS(i - 1, j) + 1 & \text{if } i, j > 0 \text{ and } x_i \neq y_j
\end{cases}
\]