The homework assignment is available at:
`http://www.jennylam.cc/courses/146-s17/homework11.html`

1. Define the longest common subsequence problem:
   - the input is two sequences seq1 = [x0, x1, ..., xn] and seq2 = [y0, y1, ..., yn];
   - the output is the length of the longest common subsequence.

   This is similar to the edit distance problem, except that we are only allowed insertions and deletions, and no substitution.

   (a) Express the problem in terms of subproblems and include the base case. (An example of how to do this is given in the notes for the recursive solution for the all-pairs shortest paths problem. The solution is also very similar to the edit distance problem. Try to write down some examples as a place to start).

   Solution. Let LCS(seq1, seq2) represent an instance of the problem. In the base case, one or both inputs are sequences of length 0, in which there is no match between the two sequences:

   $$\text{LCS(seq1, []) } = 0, \qquad \text{LCS([], seq2) } = 0.$$

   If the last characters match, then LCS(seq1, seq2) is equal to whatever the length is, for the common subsequence in the subproblem in which the last characters are removed, plus 1 for the match on the last character:

   $$\text{LCS}(\text{seq1}, \text{seq2}) = \text{LCS}(\text{seq1}[0 : n_1 - 1], \text{seq2}[0 : n_2 - 1]) + 1$$

   where n_1 and n_2 are the lengths of seq1 and se2 respectively. If the last characters don't match, we should try to remove the last character of seq1 or the last character of seq2, whichever yields the longest solution. Therefore, LCS(seq1, seq2) is equal to

   $$\min(\text{LCS}(\text{seq1}[0 : n_1 - 1], \text{seq2}), \quad \text{LCS}(\text{seq1}, \text{seq2}[0 : n_2 - 1])) \qquad \bullet$$

   (b) In preparation for a dynamic programming solution, describe:
   - the number of dimensions needed by a table which will be used to record the solutions to the subproblems
   - for each dimension of the table, what the index represents, and what its range of values is

   Solution. The problems will all be of the form LCS(seq1[0:i], seq2[0:j]) where $0 \leq n_1$ and $0 \leq n_2$. So the table will have dimensions $(n_1 + 1) \times (n_2 + 1)$. In particular, if the table is called `lcs`, then the value that should be stored in table cell `lcs[i][j]` is the value of the subproblem LCS(seq1[0:i], seq2[0:j]). •

   (c) In Java, give a dynamic programming solution to the problem (include the solution as a java file).

Solution.

```
public static int dynamicProgrammingLongestCommonSubsequence(String w1, String w2) {
   // Step 1: instantiate the table
   int n1 = w1.length();
   int n2 = w2.length();
   int[][] lcs = new int[n1+1][n2+1];

   // Step 2: base cases
   for (int i = 0; i < n1 + 1; i++)
      lcs[i][0] = 0;
   for (int j = 0; j < n2 + 1; j++)
      lcs[0][j] = 0;

   // Step 3: other cases
   for (int i = 1; i < n1 + 1; i++)
      for (int j = 1; j < n2 + 1; j++)
         lcs[i][j] = (w1.charAt(i-1) == w2.charAt(j-1)) ? lcs[i-1][j-1] + 1
               : Math.max(lcs[i-1][j], lcs[i][j-1]);

   // Step 4: return cell value of the original problem
   return lcs[n1][n2];
}
```

•

(d) What is the time complexity and space complexity of this solution? Justify your answer.

Solution. The time complexity is $O(n_1 n_2)$ since there is one for-loop which iterates $n_1 + 1$ times, followed by another which iterates $n_2 + 1$ times, followed by a nested for loop which iterates $n_1 \times n_2$ times. The space complexity is dominated by the size of the table (there are other constants for the other local variables), which is the number of cells or $O(n_1 n_2)$. •

2. Recall that the problem of making change.

(a) In preparation for a dynamic programming solution, describe:

- the number of dimensions needed by a table which will be used to record the solutions to the subproblems
- for each dimension of the table, what the index represents, and what its range of values is

Solution. Let change(v) represent the subproblem of finding the change with the input denominations (unchanged) for an amount v of money. The subproblems that will need to be calculated in order to compute change(target) will be of the form

$$\text{change(v), where } 0 \leq v \leq \text{target}$$

where target is the amount we would like to make change for. Therefore, these subproblems can be accommodated by a one-dimensional array `change[v]`, where $0 \leq v \leq$ target. •

(b) In Java, give a dynamic programming solution to the problem (include the solution as a java file).

Solution.

```
public static int DynamicProgrammingMakeChange(int[] denominations, int target) {
   int[] change = new int[target+1];
   // base case is change[0] = 0 which is already the case
   for (int v = 1; v < target + 1; v++) {
      change[v] = v;
      for (int d : denominations)
         if (d <= v && change[v] > change[v - d] + 1)
            change[v] = change[v - d] + 1;
   }
   return change[target];
}
```

•

(c) What is the time complexity and space complexity of this solution? Justify your answer.

Solution. We have a nested for-loop which iterates $O(d \cdot \text{target})$ times where d is the number of coin denominations and target is the input value. The space complexity is the size of the table which is $O(\text{target})$. •