This document will explain some general steps in solving the following dynamic programming problems:

LCS(Longest Common Subsequence)
Matrix Multiplication
Optimal Binary Search Tree
0-1 Knapsack is not on the exam.

LCS

Longest Common Subsequence consists of trying to find the longest subsequence in two words. The subsequence can be broken up by other letters. The book example demonstrates this with DNA examples.

1. First start with making a table that has one extra row and column. Write the longest sequence on the left side of the table (i), while the shortest on the top (j).
2. Initialize the extra row and column with 0's.
3. Look at each (i,j) combination: starting from the first non-initialized row and column. Ask yourself this question:
    1. Do the two characters match?
        1. If YES: Place a 1 and add the item that is (i-1, j-1) [As in, take the Upper Left or NW square's value). Draw an arrow pointing to the (i-1, j-1)th square.
        2. If NO:
            1. Compare (i, j-1) and (i-1, j). Take the highest one and draw an arrow to the chosen one. In other words, look at the square directly above and to the left of the current spot. Take the highest. Remember to draw the arrow.
4. Repeat step 3 until the last index combination is computed. The number in the lowest right hand corner is the length of the LCS.
5. To calculate the contents of the LCS, simply follow the arrows back up until the initialization row is reached. Each time a diagonal arrow is found (where the characters matched), write that down as it is a common character in the LCS.

Matrix Multiplication

I find matrix multiplication easier done if the initialization steps are done prior to filling in the table, and then following the algorithm for the remaining.

1. Write down the p table; starting with p(0) up to p(n).
2. Draw a diagonal table, as displayed in the book.
3. Initialize all (i,i) to 0. (We can call this the bottom row).
4. The next row (which is 1 item smaller than the bottom row) is simply the first combination of multiplications. For example, p(0)*p(1)*p(2) would be in the first column in this row. The second column would contain P(1)*p(2)*p(3) and so on. The reason why this is when the equation is calculated to find the minimum, the (i,j) combinations will equal: (i,k), (k+1,j), which, will be: (i,i) and (j,j). In step 3, we set all these to 0. Thusly, we will end up with: 0 + 0 + p(i-1)*p(k)*p(j). s[i,j] will be i in this case.
5. For the remainder rows (third row and up), I find using the equation in the book easiest. It is a lot of work, but is easy to follow along. For each column (j) in the third row (i), compute the results of the equation: min(i<=k<j: m[i,k] + m[k+1,j] + p(i-1)*p(k)*p(j)). This equation states that we will find the minimum value of the statement from a list of computations. The lowest will be chosen. It will start at i and go up to (j-1). k will be incremented each time through this list. i and j will be remain the same. The only time i and j will change is when: a new row is looked at and when a new column is looked at. s[i,j] will be whatever is the smallest amount of multiplications. For example, if k = 2 turns

out to be the smallest amount, then we will set s[i,j] to be 2.

6. Continue step 5 until the table is completed. The top value in the diagonal table is the minimum amount of computations needed.
7. To compute the ()'s, there is a simple recursive function on page 338. It is a simple matter (but some work) going through the s table with the indicated values.

Optimal Binary Search Trees

As with matrix multiplication, I find using the equations in the book the easiest. This problem has three diagonal tables: an e, w and root table. The e table is the cost of the probabilities for one of the best optimal trees. The w table is the probability table. This table is used to assist in calculating the probabilities for e. The root table is structure of the optimal binary search tree. The general process will involve: setting a value for w, then for e, and finally for root.

1. Write down the probabilities in a table. The first row will be p and the second will be q. p starts at index 1 and q starts at index 0 and goes up to N.
2. For table e and w, initialize each bottom row with the probabilities indicated by q(i). For the left side, j, start 0 and go up to N (5 in the book). For the right side, i, start at 1 and go to N+1. For table root, initialize the bottom row with starting at 1 and incrementing until N.
3. Walk across the row, like in matrix multiplication: Calculate w: w[i,j] = w[i, j-1] + p(j) + q(j).
4. Like in matrix multiplication, we need to find the minimum amount. In this case it is probabilities. We will execute the following statement: min(i<=r<=j: e[i,r-1] + e[r+1, j] + w[i,j]).
5. Choose the lowest and then place that value in e[i,j].
6. Take the r and place it in root[i,j]. This is just like with k in matrix multiplication.
7. Repeat steps 3-6 until the tables are completed. The top value in the diagonal e table is the lowest probability.
8. The algorithm for computing the Optimal BST from the root table is left for homework in the book. This is the algorithm (I gathered this from somewhere else):

```
CONSTRUCT-OPTIMAL-BST(root)
r ← root[1, m]
print r is the root
CONSTRUCT-OPT-SUBTREE(1, r – 1, r, “left”, root)
CONSTRUCT-OPT-SUBTREE(r + 1, m, r, “right”, root)

CONSTRUCT-OPT-SUBTREE(i, j, r, childtype, root)
if(i > j)
print d[j] is the childtype child of r
else
print k[root[i, j]] is the childtype child of r
CONSTRUCT-OPT-SUBTREE(i, k[root[i, j] - 1], k[root[i, j]], left, root)
CONSTRUCT-OPT-SUBTREE(k[root[i, j] + 1], j, k[root[i, j]], right, root)
```