Problem 1

We can recycle the LCS algorithm from class. However, we will compare the set $X^i = \{x_1, \ldots, x_{n-1}\}$ to $X^j = \{x_2, \ldots, x_n\}$, $X^i, X^j \subset X$. Also, we will not not allow $i \geq j$, to make sure that we are only checking later terms against previous terms. This means that we will only fill out the upper triangular elements of b and c. Instead of searching for where elements are equal, we want their absolute difference is less than or equal to 5. The printing code in the end is also very similar to the printing routine presented in the book, with one minor exception. We need to print out not only the elements x_j which satisfied the constraint, but when we are done we need the element x_i which was compared to the initial x_j. I took care of this issue by updating a variable called first with an x_i element.

Algorithm

```
/* Assume that indexing of c starts at 0, and that indexing of all other arrays start at 1. (I think this makes
this code easier to read, because otherwise the indices of c are shifted to accomodate the zeros on the border of
the array.)*/
ProblemOne(X):
    n = X.length()
    b = Allocate memory for an n x n character array
    c = zeros(0:n; 0:n) Gives an n+1 x n+1 array of zeros
    for i = 1:n-1
        for j = 2:n
            if ( | x[i] - x[j] | <= 5):
                c[i][j] = c[i-1][j-1] + 1
                b[i][j] = ↖
            else if (c[i-1][j]) >= c[i][j-1]):
                c[i][j] = c[i-1][j]
                b[i][j] = ↑
            else:
                c[i][j] = c[i][j-1]
                b[i][j] = ←
    return c, b

Retrieve-LCS(b, X, i, j)
    if (( i == 0) || (j == 0)):
        Print-LCS(sequence, first)
    if (b[i][j] == ↖ ):
        Retrieve-LCS(b, x, i-1, j-1)
        sequence.push(x[j])
        first = x[i]

Print-LCS(sequence, first):
    print (first)
    for x in sequence.length() :
        print (sequence.pop())
```

Time Complexity

This algorithm will perform half of the work or the algorithm presented in class for an LCS. The runtime is still $O(n^2)$.The printing will require $O(n)$ time, as stated in the textbook.

Problem 2

The Catalan Number $\mathcal{C}_{n-1} = \frac{1}{n}\binom{2(n-1)}{n-1}$ gives the number of ways to parenthesize n matrices. To calculate the cost of multiplying an $p_1 \times p_2$ matrix by an $p_2 \times p_3$ matrix, we have to calculate $p_1 p_2 p_3$, which requires two products. We have to calculate $n-1$ of these products to compute the cost of multiplying n matrices, since

during each MM operation we reduce the number of matrices by one, until we get to one left. So, we have to perform $2 * (n-1) * \frac{1}{n} * \frac{2n!}{n!n!}$ multiplications. Finally, we have to do a reduction on the set of the chains' costs. Since there are $\mathcal{C}_n$ chains, and we need to check the cost of each chain once as we perform the the reduction, we are adding $\mathcal{C}_n$ operations.
The recursive solution will be asymptotically faster, because it calculates all of the possible sub problems for any problem, but in this way does not have to

Problem 3

The result is not always optimal. Consider the multiplication of matrices corresponding to the 'dimension vector' $\vec{p} = [2, 3, 2, 1]$. The proposed greedy method says to parenthesize the first and second matrices, since $k = 2$. This results in $2(3(2)) + 2(2(1)) = 12 + 4$ scalar products. The optimal parenthization is at $k = 1$, giving $2(3(1)) + 3(2(1)) = 12$ scalar products.

Problem 4

Algorithm

```
// Indexing starts at 1, not at zero. array[0] gives a seg-fault.
m = X.length()
n = Y.length()
c = zeros(1:m; 1:n)

LCSRecursion(i, j, X, Y):
    if (i == 0)||(j == 0): // The string has been entirely investigated.
        return 0
    if(c[i][j] != 0): // We have already calculated this.
        return c[i][j]
    if(x[i] == y[j]): // We have a match!
        c[i][j] = LCSRecursion(i-1, j-1, X, Y) + 1
        return c[i][j]
    else: //return the larger of the LCSs between the indicated substrings.
        len1 = LCSRecursion(i-1, j, X, Y)
        len2 = LCSRecursion(i, j-1, X, Y)
        if (len1 ≥ len2):
            c[i][j] = len1
            return c[i][j]
        else:
            c[i][j] = len2
            return c[i][j]
end

lcs = LCSRecursion(m, n, X, Y)
```

Time Complexity

If one string is a substring of another, we will have minimal runtime. This is because every call to the function will have a match and we will have no branching. Therefore, we can give the lower bound as:

$$\Omega(M), \quad M = \min\{m, n\}$$