Problem 1.

a. The filled table is provided below.

i\D	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	4	4	4	4	4	4	4	4	4
2	0	0	4	4	7	7	7	7	7	7	7
3	0	0	4	6	7	10	10	13	13	13	13
4	0	0	4	6	7	10	10	13	13	15	15
5	0	0	4	6	7	10	10	13	13	15	17

Applying the `PrintOptKnapsack` procedure described in the lecture notes, we find items 1, 3, and 5 are in the optimal solution. Their weights are 2, 3, and 5 and their values are 4, 6, and 7. Hence the total weight is 10 and the total value is 17.

b. We find that ACD is the LCS. The filled table is provided below. The numbers that are bolded represent the cells that the `PrintLCS` procedure goes through. The numbers that are red represent cells where we print X_i.

	j	0	1	2	3	4	5
i		Y_j	A	C	D	E	F
0	X_i	0	0	**0**	0	0	0
1	A	0	1	1	**1**	1	1
2	B	0	1	1	**1**	1	1
3	A	0	1	1	**1**	1	1
4	E	0	1	1	**1**	2	2
5	C	0	1	2	2	**2**	2
6	C	0	1	2	2	**2**	2
7	D	0	1	2	3	3	**3**

c. We have $(d_0, d_1, d_2, d_3, d_4, d_5) = (1, 5, 2, 6, 1, 4)$. The minimum number of multiplications is 28. This is achieved via $((A \times B) \times (C \times D)) \times E$. The filled M-matrix and S-matrix are shown below.

M-matrix

		i				
		1	2	3	4	5
	5	28	42	20	42	0
	4	24	22	12	0	
j	3	22	60	0		
	2	10	0			
	1	0				

S-matrix

		i				
		1	2	3	4	5
	5	4	4	4	4	$\perp$
	4	2	2	3	$\perp$	
j	3	2	2	$\perp$		
	2	1	$\perp$			
	1	$\perp$				

Problem 2.

- We consider the algorithm presented in problem 2 of problem set #5. After running the algorithm, we have $A = \{0, 7, 8, 13, 17, 24\}$ and $S = \{\text{N/A}, 1, 1, 3, 3, 5\}$.
  - We show how we fill the cell at $A[5]$. At this point, we will have $A = \{0, 7, 8, 13\}$ and $S = \{\text{N/A}, 1, 1, 3\}$. Let $i = 5$. We first set $A[i] = \infty$. We then iterate from $j = 1$ to $i - 1 = 4$. We attempt to find the lowest value out of $A[j] + C[j][5]$. We have
    * $j = 1 \longrightarrow A[1] + C[1][5] = 0 + 21 = 21$,
    * $j = 2 \longrightarrow A[2] + C[2][5] = 7 + 13 = 20$,
    * $j = 3 \longrightarrow A[3] + C[3][5] = 8 + 9 = 17$,
    * $j = 4 \longrightarrow A[4] + C[4][5] = 13 + 8 = 21$.

    We see we get the lowest value when $j = 3$. As a result, we set $A[i] = 17$ and $S[i] = 3$ and have $A = \{0, 7, 8, 13, 17\}$ and $S = \{\text{N/A}, 1, 1, 3, 3\}$ after this step.
  - We now show how we fill the cell at $A[6]$. Let $i = 6$. We first set $A[i] = \infty$. We then iterate from $j = 1$ to $i - 1 = 5$. We attempt to find the lowest value out of $A[j] + C[j][6]$. We have
    * $j = 1 \longrightarrow A[1] + C[1][6] = 0 + \infty = \infty$,
    * $j = 2 \longrightarrow A[2] + C[2][6] = 7 + \infty = \infty$,
    * $j = 3 \longrightarrow A[3] + C[3][6] = 8 + 17 = 25$,
    * $j = 4 \longrightarrow A[4] + C[4][6] = 13 + 15 = 28$,
    * $j = 5 \longrightarrow A[5] + C[5][6] = 17 + 7 = 24$.

    We see we get the lowest value when $j = 5$. As a result, we set $A[i] = 24$ and $S[i] = 5$ and have $A = \{0, 7, 8, 13, 17, 24\}$ and $S = \{\text{N/A}, 1, 1, 3, 3, 5\}$ after this step.
- $S = \{\text{N/A}, 1, 1, 3, 3, 5\}$, so renting from 1 to 3, 3 to 5, and 5 to 6 yields the minimum cost.

Problem 3.

a. The general idea of our algorithm is to apply the `LCS` algorithm discussed in class. However, instead of using two different strings, we will simply call the `LCS` algorithm with some string X and the reverse of X. Let us define $Y = \langle x_n \dots x_1 \rangle = \langle y_1 \dots y_m \rangle$ as the reverse of X. We get the recursion

$$\text{LSP}(\langle x_1 \dots x_n \rangle,\ \langle y_1 \dots y_m \rangle) = \max \begin{Bmatrix} \text{LSP}(\langle x_1 \dots x_{n-1} \rangle,\ \langle y_1 \dots y_m \rangle), \\ \text{LSP}(\langle x_1 \dots x_n \rangle,\ \langle y_1 \dots y_{m-1} \rangle), \\ (\text{if } x_n = y_m)\ 1 + \text{LSP}(\langle x_1 \dots x_{n-1} \rangle,\ \langle y_1 \dots y_{m-1} \rangle). \end{Bmatrix}$$

All recursive calls live in a small domain: the first i characters of X and the first j characters of Y.

b. We define $D[i][j]$ to hold the result of the (i, j)-recursion call. For each $0 \le i \le n$ and $0 \le j \le n$ (where n is the length of the input string X), $D[i][j]$ is the length of the longest common subsequence of $\langle x_1 \dots x_n \rangle$ and $\langle x_n \dots x_1 \rangle$. We have

$$D[i][j] = \max \begin{Bmatrix} D[i-1][j], \\ D[i][j-1], \\ (\text{if } x_i = y_j)\ 1 + D[i-1][j-1]. \end{Bmatrix}$$

The dimension of the table will be $n \times n$. The cell at $D[n][n]$ will store the value of an optimal solution.

c. The pseudocode for `LongestSubPalindrome` is provided below. It prints and returns the length of the longest sub-palindrome. Note that it uses the `LCS` algorithm discussed in the lecture slides as a helper function. The code is written in Lua. Note that the `local` keyword is used to create a new variable. `{}` is used to create a new array. Other Lua-specific functions are explained in comments throughout the code.

```lua
function LCS(X, Y)
    local n = X:len() -- Get the length of X
    local m = Y:len() -- Get the length of Y

    local D = {}

    for i = 0, n do
        D[i] = {}

        for j = 0, m do
            D[i][j] = 0
        end
```

```
    end

    for i = 1, n do
        for j = 1, m do
            D[i][j] = D[i - 1][j]

            if D[i][j - 1] > D[i][j] then
                D[i][j] = D[i][j - 1]
            end

            -- sub(i, i) gets the ith character of X
            -- sub(j, j) gets the jth character of Y
            if X:sub(i, i) == Y:sub(j, j) and D[i - 1][j - 1] + 1 > D[i][j] then
                D[i][j] = D[i - 1][j - 1] + 1
            end
        end
    end

    return D
end

function LongestSubPalindrome(X)
    local n = X:len() -- Get the length of X
    local m = n
    local D = LCS(X, X:reverse()) -- reverse() reverses X
    print(D[n][m])
    return D
end
```

As discussed in the lecture notes, `LCS` runs in $O(n+m+nm)$ time where n is the length of the first string and m is the length of the second string. Our algorithm `LongestSubPalindrome` calls `LCS` with some string X and the reverse of X. If X has length n, then the reverse of X also has length n. This means we are calling `LCS` with two strings of length n meaning we can conclude the running time is in $O(2n + n^2) \in O(n^2)$.

d. The pseudocode for `PrintLongestSubPalindrome` is provided below.

```
function PrintLongestSubPalindrome(D, i, j, X)
    if i > 0 and j > 0 then
        if D[i][j] == D[i - 1][j - 1] + 1 then
            PrintLongestSubPalindrome(D, i - 1, j - 1, X)
            print(X:sub(i, i))
        elseif D[i][j] == D[i - 1][j] then
            PrintLongestSubPalindrome(D, i - 1, j, X)
        else
            PrintLongestSubPalindrome(D, i, j - 1, X)
        end
```

```
        end
end
```

Notice that the algorithm is almost exactly the same as the algorithm derived for `PrintLCS` in the lecture notes. The algorithm travels at most n cells up and n cells left in D. As a result, we can conclude the running time is in $O(n)$.

e. (i) The filled table is shown below.

	j	0	1	2	3	4	5	6
i		Y_j	e	d	q	q	e	q
0	X_i	0	0	**0**	0	0	0	0
1	p	0	0	**0**	0	0	0	1
2	e	0	1	1	**1**	1	1	1
3	q	0	1	1	2	**2**	2	2
4	q	0	1	1	2	3	**3**	3
5	d	0	1	2	2	3	**3**	3
6	e	0	1	2	2	3	4	**4**

(ii) The sub-palindrome generated is `eqqe`. We first used the `LongestSubPalindrome` algorithm to generate D, then used the `PrintLongestSubPalindrome` algorithm with D, $i = 6$, $j = 6$, and $X =$ `"peqqde"` to output the letters of the palindrome.