15.2-2

As observation, it must be said that naïve recursive implementation just returns the product of MATRIX-CHAIN MULTIPLY between i and computed s[i,j] AND the counterpart from s[i,j]+1 and j).

It must also be said that "base" cases can defined in more ways (e.g. when i=j, i+1=j and so forth).

Putting these together, the naïve recursive implementation can look like below:

```
MATRIX-CHAIN-MULTIPLY(A, s, i, j)
        if i == j then return A[i]
        endif
        if j==i+1 then return A[i]*A[j]
        endif
//else
return
MATRIX-CHAIN-MULTIPLY(A,s,i,s[i,j])*MATRIX-CHAIN-MULTIPLY(A,s,s[i, j]+1,j)
```

15.2-3

We prove using structural induction method the statement q(n): $p(n) \geq t2^n$ $\forall n \geq 1$, where t is a positive constant (note that would be equivalent with the definition of Big-Omega)

The base case is not important, we can assume it solved.

For the general step, assuming that $p(c) \geq t2^c$ for all $c<n$ (induction hypothesis).

We must prove that $p(n) \geq t2^n$

Using the recurrence and the induction hypothesis for k and n-k (since both are <n):

$P(n) \geq \sum_{k=1}^{n-1} (t2^k t2^{n-k}) = \sum_{k=1}^{n-1} (t^2 2^n) = (n-1)t^2 2^n$

Now we verify whether $(n-1)t^2 2^n \geq t2^n$ which is equivalent with $(n-1)t \geq 1$ and this obviously holds (we can adjust the constant t such that (n-1) multiplied by it remains at least equal to 1).

In conclusion, we have proved that q(k) and q(n-k) => q(n), therefore $p(n) \geq t2^n$ $\forall n \geq 1$, which means that $p(n)$ =Big-Omega(2^n) as required

15.2-4

For this problem, the vertices of the graph are just all ordered pairs (i,j) where $1 \le i \le j \le n$.

This means that we can compute the no. of vertices using double summation like below.

$$\sum_{i=1}^{n}\sum_{j=i}^{n} 1 = \sum_{i=1}^{n}(n-i+1) = \sum_{i=1}^{n} n - \sum_{i=1}^{n} i + \sum_{i=1}^{n} 1 = n\sum_{i=1}^{n} 1 - \sum_{i=1}^{n} i + \sum_{i=1}^{n} 1 =$$

$$= n^2 - n(n+1)/2 + n = n^2 - n^2/2 - n/2 + n = n^2/2 + n/2 = n(n+1)/2$$

In this context, every subproblem V_{ij} has precisely j-i subproblems that must be solved.

The edges suggest that, in order to optimally solve parenthesization for matrix product between i and j, it is needed to first optimally parenthesize the product between i and k and k and j respectively (where k is any other intermediary potential splitting point from the path).

Therefore, we obtain a number of edges which is equal to

$$\sum_{i=1}^{n}\sum_{j=i}^{n}(j-i) = \sum_{i=1}^{n}\left(\sum_{j=i}^{n} j - \sum_{j=i}^{n} i\right) = \sum_{i=1}^{n}\left(\sum_{j=1}^{n} j - \sum_{j=1}^{i-1} j - i\sum_{j=i}^{n} 1\right) =$$

$$= \sum_{i=1}^{n}\left(\frac{n(n+1)}{2} - \frac{i(i-1)}{2} - i(n-i+1)\right) = \frac{n^2(n+1)}{2} - \frac{1}{2}\sum_{i=1}^{n} i^2 + \frac{1}{2}\sum_{i=1}^{n} i - n\sum_{i=1}^{n} i + \sum_{i=1}^{n} i^2 - \sum_{i=1}^{n} i =$$

$$= \frac{n^2(n+1)}{2} + \frac{1}{2}\sum_{i=1}^{n} i^2 - (n+1/2)\sum_{i=1}^{n} i = \frac{n^2(n+1)}{2} + \frac{n(n+1)(2n+1)}{12} - \frac{n^2(n+1)}{2} - \frac{n(n+1)}{4} = \frac{n(n+1)(2n+1-3)}{12} =$$

$$= \frac{(n-1)n(n+1)}{6} = \frac{n^3-n}{6}$$

15.2-6

The most natural way is to use induction by the number of matrices.

We prove using structural induction p(n): a full parenthetization of an n-element expression has exactly n-1 pairs of parentheses for any $n \ge 1$.

Base case: If n=1 , then there are needed 1-1=0 pairs of parentheses, which is true, because a single matrix does not need any parenthesis.

Inductive step: We assume as induction hypothesis that a full parenthesization of some k-element expression needs k-1 pairs of parenthesese, where k<n.

We must prove that a full parenthesization of an n-element expression needs n-1 pairs of parentheses.

When we have an n-element expression, this can be split by at least one point, let's say c. It is obvious that c and n-c are both <n, so we can apply the induction hypothesis for the two chunks (up to c and from c+1 to n).

According to the induction hypothesis, there are needed (c-1) +[(n-c-1+1)-1]=(c-1)+(n-c-1)=n-2 pairs of parentheses. Now, adding the pair for the whole expression (which gathers the two chunks), we obtain n-2+1=n-1, which proves the induction claim.
Since we have proved p(n) holds, the induction proof is now complete and p(n) is true for any n greater than or equal to 1.

15.4-2

We can only use the c table.
In order to reconstruct the optimal LCS, we compare two characters of the original sequences X and Y. If they are identical, then we call recursively the algorithm for i-1 and j-1 (previous characters). If they are not identical, then we compare the values from previous row and previous column from table c and, based on the one which has a bigger value, we recursively call the algorithm in either direction (e.g. if c[i-1,j] is the bigger, then we call it for i-1 and j). In pseudo-code, this can be written like below:

```
DISPLAY-LCS(table c, X, Y, i,j)
If c[i,j]==0 then return –
If X[i]==Y[j] then DISPLAY-LCS(table c, X, Y, i-1, j-1)
                        Print X[i] //or Y[j] as well
                Else if c[i-1,j] < c[i,j-1] then return DISPLAY-LCS(table c, X, Y, i, j-1)
                        Else return DISPLAY-LCS(table c, X, Y, i-1, j)
```

15.4-3

```
Memoized-LCS(i,j)
        If LCS(i,j) is not yet calculated then
                If i==0 OR j==0 then LCS(i,j)=0
        Else //if last characters are the same
                If X[i]==Y[j] then LCS(i,j) =Memoized-LCS(i-1,j-1)+1
                        Else //the last characters are not the same
                        LCS(i,j) =Max(Memoized-LCS(i-1,j), Memoized-LCS(i,j-1))
        Endif
```

Return LCS(i,j) //if it was calculated already, then return the stored value

Complexity study – the above memorized version has O(mn) running time because i has m+1 potential values and j has n+1 potential values. When the value is already calculated, then it is returned in O(1) time. When the value is not computed, then it is returned after performing some constant number of calls. In conclusion, the running time is O(mn) because it is equal with the necessary time to compute all values.

15.4-5

Note the algorithm must return the longest subsequence and not its length only. Therefore, we need to maintain a secondary array for storing the longest monotonical subsequence and to reconstruct it in the end based on the saved position.

Assuming the array length is used to compute the max length, we take the array pos to keep track of the position where the longest subsequence occurs. Also, we assume the original sequence of n numbers is s. In the end, when outputting the longest subsequence, we need to call it in reverse (because we proceed backwards).

```
For i=0 to n-1 do
    For j=0 to i-1 do
        If (s[j]<s[i]) AND (length[i]<length[j]+1) then
                length[i]=length[j]+1
                pos[i]=j
max=length[0], index=0 //initializations
for i=1 to n-1 do
    if (length[i]<max) then max=length[i], index=i

while (index is NOT -1) do //reconstruction proceeding backwards
    push.back(s[index])
    pos=pos[index]
reverse(subsequence)
return subsequence
```

Complexity study – the running time of the algorithm is dominated by the number of iterations of the inner loop, which is $\mathbf{\sum_{i=0}^{n-1}\sum_{j=0}^{i-1} 1 = \sum_{i=0}^{n-1} i = n(n-1)/2 = O(n^2)}$ as required (inside the loop all operations require constant time, thus the running time of the algorithm is proportional with the number of iterations).